Sistemas alternativos de interconexión: El IEEE1284 en Linux

Jose Renau Ardevol

renau@@acm.org

20 de diciembre de 2003

Resumen

En la realización del proyecto "sistemas alternativos de interconexión: *El IEEE1284 en Linux*", se ha implementado el estándar IEEE1284 en Linux. Este estándar especifica unas características y prestaciones que han de cumplir los puertos paralelos.

En la implementación del IEEE1284 en Linux se ha creado una nueva API gestora del puerto paralelo que soporta los nuevos modos de operación. Se han modificado diversos dispositivos para que se adapten a la nueva API. Utilizando los nuevos modos, ECP y EPP principalmente, se consigue aumentar las prestaciones, así aprovechar la especificación PnP de Microsoft sobre los puertos paralelos.

El nuevo software desarrollado se encuentra en la distribución del Kernel de Linux desde la versión 2.1.33.

El proyecto está íntimamente ligado al proyecto "Sistemas alternativos de interconexión: *Networking with SCSI*" puesto que ambos proyectos comparten código en la parte de gestión de red. No obstante mientras que un proyecto utiliza el bus SCSI como sistema de interconexión, el proyecto presente utiliza las nuevas capacidades del puerto paralelo como sistema de interconexión.

Índice general

I. Introducción 10

1. Introducción 11
1.1. Motivaciones 11
1.2. Objetivos alcanzados 12
1.3. Organización de la documentación 12

II. Conocimientos Básicos 14

2. Las redes TCP/IP 15
2.1. Introducción 15
2.1.1. Organización del capítulo 16
2.2. Estructura del protocolo 17
2.2.1. Arquitectura por niveles 17
2.2.2. Encapsulamiento 18
2.2.3. Ejemplo ilustrativo 20
2.2.4. Conceptos de *routing* 21
2.3. El nivel de enlace 24
2.4. El protocolo ARP/RARP 25
2.5. El nivel red 28
2.5.1. Los paquetes IP 29
2.5.2. Máscaras y direcciones 30
2.5.3. El enrutamiento 31
2.5.4. IPv6 32
2.6. El nivel de transporte 34
2.6.1. El protocolo UDP 34
2.6.2. El protocolo TCP 34
2.7. Nivel de aplicación 37
2.7.1. Aplicaciones basadas en UDP 37
2.7.2. Aplicaciones basadas en TCP 38
2.7.3. Aplicaciones basadas en otros protocolos 38

3. Linux - El sistema operativo **40**
3.1. Introducción . . . 40
3.1.1. Organización del capítulo . . . 40
3.1.2. Principales características . . . 41
3.2. El Kernel . . . 43
3.2.1. Normas de codificación . . . 43
3.2.2. Versiones de Kernel . . . 44
3.2.3. Organización del Kernel . . . 45
3.2.4. Estructuras más comunes . . . 47
3.2.5. Compilación . . . 51
3.3. Los Modules . . . 55
3.3.1. Que son los modules . . . 55
3.3.2. Otras aproximaciones . . . 55
3.3.3. Como se gestionan . . . 57
3.3.4. Filosofia de trabajo . . . 57
3.3.5. Implementación . . . 58
3.3.6. Carga y descarga por parte del kernel . . . 60
3.3.7. Conclusiones . . . 61
3.4. El sistema de ficheros /proc . . . 62
3.4.1. Principales entradas . . . 62
3.4.2. Creación de una entrada . . . 63
3.5. La Red . . . 65
3.5.1. La estructura sk_buff . . . 66
3.5.2. La estructura device . . . 67
3.5.3. Diseño de Devices de Red . . . 70
3.6. El subsistema SCSI . . . 75
3.6.1. Introducción . . . 75
3.6.2. Drivers de controladoras . . . 76
3.6.3. Inicialización del nivel bajo . . . 79
3.6.4. El núcleo del sistema . . . 80
3.6.5. Drivers de dispositivos . . . 82
3.6.6. Inicialización del sistema . . . 85

4. Puerto Paralelo **88**
4.1. Introducción . . . 88
4.2. Conceptos Básicos . . . 90
4.2.1. Las señales . . . 90
4.2.2. Nomenclatura . . . 90
4.2.3. Registro de datos . . . 92
4.2.4. Registro status . . . 92
4.2.5. Registro de control . . . 93
4.2.6. Detección . . . 94
4.3. Modo PS2 . . . 95
4.3.1. Detección . . . 95

4.3.2. Operación . . . 95
4.4. Modo EPP . . . 97
4.4.1. Detección . . . 97
4.4.2. Operación . . . 99
4.5. Modo ECP . . . 101
4.5.1. FIFO . . . 101
4.5.2. Registros . . . 102
4.5.3. Compresión . . . 103
4.5.4. DMA . . . 104
4.5.5. Detección . . . 105
4.5.6. Operación . . . 107

5. Sistemas de interconexión **109**
5.1. Introducción . . . 109
5.2. Myrinet . . . 110
5.3. SCI . . . 111
5.4. USB . . . 112
5.5. ATM . . . 113

III. Implementación del proyecto 115

6. Objetivos y estructuración **116**
6.1. Introducción . . . 116
6.2. Objetivos . . . 117
6.3. Filosofía de diseño . . . 119

7. parport **122**
7.1. Problemática encontrada . . . 122
7.2. Características deseadas . . . 123
7.3. Diseño . . . 125
7.3.1. Estructuras definidas . . . 125
7.3.2. Especificación de la API . . . 128
7.3.3. Module . . . 130
7.3.4. Interacción con Proc FileSystem . . . 131
7.4. Implementación . . . 134
7.4.1. Activación . . . 134
7.4.2. Ficheros implicados . . . 134
7.5. Absorción de subsistemas . . . 136
7.5.1. Parámetros de configuración . . . 136
7.5.2. Registrarse en el parport . . . 137
7.5.3. Adquisición del control del puerto . . . 138
7.6. Conclusiones . . . 140

8. Threaded Device Support **141**
8.1. Introducción . . . 141
8.2. Características deseadas . . . 141
8.3. Diseño . . . 144
8.3.1. Estructuras definidas . . . 144
8.3.2. Especificación de la API . . . 146
8.4. Modificaciones en el subsistema de red . . . 148
8.5. Implementación . . . 150
8.5.1. Activación . . . 150
8.5.2. Ficheros implicados . . . 151
8.6. Funcionamiento . . . 152
8.6.1. Enviar Mensaje . . . 152
8.6.2. Recibir mensaje . . . 153
8.7. Conclusiones . . . 156

9. EPLIP **157**
9.1. Introducción . . . 157
9.2. Características deseadas . . . 157
9.3. Diseño . . . 159
9.3.1. Estructuras . . . 159
9.3.2. Inicialización . . . 160
9.4. Funcionamiento . . . 163
9.4.1. Activación del dispositivo . . . 163
9.4.2. Envío y recepción de paquetes . . . 166
9.5. Conclusiones . . . 169

10. Conclusiones y valoración **170**
10.1. Evolución . . . 170
10.2. Las interioridades del núcleo . . . 171
10.3. El comportamiento del puerto paralelo . . . 171
10.4. Resultados preliminares . . . 172
10.4.1. Latencias . . . 172
10.4.2. *Bandwidth* . . . 173
10.5. Conclusiones del proyecto . . . 175

IV. Apéndices **176**

11. The GNU General Public License **177**
11.1. Preamble . . . 177
11.2. Terms and Conditions . . . 178
11.3. How to Apply These Terms . . . 182

Índice de figuras

2.1. Capas del TCP/IP 18
2.2. Correspondencia entre TCP/IP y OSI 19
2.3. Sistema de encapsulamiento por niveles 20
2.4. Trama Ethernet 20
2.5. Encapsulación usada en la transmisión FTP 22
2.6. Aspecto de una cabecera Ethernet 24
2.7. Aspecto de trama ARP 26
2.8. Aspecto de un paquete IP 29
2.9. Aspecto de un paquete UDP 34
2.10. Aspecto de un paquete TCP 35

3.1. Configuración mediante xconfig 52
3.2. Estructura parcial de módulos de red 65
3.3. Secuencia de llamadas al device de red para realizar un ping 74
3.4. Estructura del subsistema SCSI de Linux 76
3.5. Ejemplo de configuración de listas 80
3.6. Sistema de compartición de dispositivos proporcionado por el núcleo del sistema SCSI, a través de las estructuras *Scsi_Device*. 84

6.1. Esquema estructural de niveles 120
6.2. Ejemplo de la estructura en una configuración de máquina con tarjetas SCSI, puerto paralelo y SCI 121

7.1. Relación entre parport y ppd 126

8.1. Posicionamiento del TDS dentro de Linux 143
8.2. Estructura del espacio de memoria devuelto por *dev_alloc_skb()*. 148
8.3. Espacio total reservado para datos más *skb* 149
8.4. Ubicación de los datos dentro del espacio reservado 149
8.5. Estado de la memoria después de la función *NEW_alloc_skb()*. 150
8.6. Aspecto de un paquete+*sk_buff* preparado para el kernel de red. 150
8.7. Transmisión de un mensaje 153
8.8. Recepción de un mensaje 154

9.1. Formato de un paquete EPLIP con su cabecera. 160

9.2. Relaciones entre TDS, device y eplip_net 162
9.3. Secuencia de llamadas entre niveles para abrir un dispositivo de red. . . . 164
9.4. Secuencia de llamadas entre niveles para la recepción de paquetes 168

Índice de cuadros

2.1. Ejemplos de MTU 26
2.2. Clases IP 30

3.1. Directorios a un nivel de profundidad 45
3.2. Estructura del comando scsi (*Scsi_Cmnd*) 81
3.3. Estructura del dispositivo scsi (*Scsi_Device*) 83

4.1. Señales del DB25 91
4.2. Registros del puerto paralelo 91
4.3. Mapeo de los registros 92
4.4. Registros en el modo EPP 98

10.1. Comparativa de latencias desde diferentes niveles de red 173
10.2. Comparativa de *bandwidth* con *bw_tcp*. 174

Parte I.

Introducción

Capítulo 1 ❑

Introducción

1.1. Motivaciones

Este proyecto se originó hace más de un año cuando surgió la necesidad de crear una red de ordenadores con sistemas de conexión alternativos. Primero se pensó en redes SCI, no obstante, por problemas de disponibilidad fue imposible realizarlo.

Al buscar alternativas en la creación de un multicomputador se pensó en utilizar placas SCSI. Las placas SCSI poseen un coste reducido y pueden alcanzar anchos de banda aceptables, entre 10Mbytes/s y 40Mbytes/s.

Simultaneamente se observó que un proyecto de la Universidad de Pardue utilizaba placas con puertos paralelo para la creación de un multicomputador. En esa versión consiguen muy buenas latencias ($3uS$), aunque no conseguían un ancho de banda apropiado ($65KBytes/s$). No obstante, en Pardue, utilizan puertos paralelos sin aprovechar los nuevos modos de transferecia por lo que es factible aumentar el ancho de banda.

A partir de los recursos disponibles se empezaron dos ramas de trabajo altamente ligadas:

- Crear una red de ordenadores usando placas SCSI adaptec 1542C, integrandolo como mínimo con protocolos TCP/IP.

- Utilizar los nuevos modos del puerto paralelo como sistema de interconexión entre ordenadores.

El soporte escogido para la implementación de todo el proyecto fue el sistema operativo Linux, por la disponibilidad de los fuentes, gran soporte de programas, conocimientos previos en la programación del núcleo, posibilidad de contactar con gente para la expansión del proyecto en otro tipo de plataformas, posibilidad de su integración en su distribución básica ...

Al empezar a estudiar más profundamente el Linux se observó que el Kernel tenía muchas carencias en la gestión y utilización de los puertos paralelos. Por este motivo se ampliaron los objetivos y se decidió crear un nuevo módulo gestor del puerto paralelo que soportase todas las nuevas prestaciones que han surgido y no estaban soportadas.

1.2. Objetivos alcanzados

En la realización del proyecto se han creado dos grupos altamente ligados. Por una lado se ha creado el driver SCSI con soporte a red, mientras que por otro utiliza el puerto paralelo como sistema de interconexión. Estos dos proyectos comparten gran parte de código encargado de la gestión de redes.

En el proyecto encargado del puerto paralelo se optó por potenciar un nuevo módulo gestor con la intención de soportar el nuevo estándar IEEE1284. El nuevo módulo de Linux, llamado `parport`, detecta puertos siguiendo las especificaciones PnP de Microsoft y cumple con la especificación del IEEE.

Una vez empezó a estar operativo el módulo gestor del puerto paralelo se discutió en la mailing list linux-parport@@tornet.net que API tenía que proporcionar [1]. El módulo proporciona detección de los nuevos modos del puerto paralelo (ECP,EPP, ...), también permite la compartición de un mismo puerto paralelo por diversos dispositivos.

Se ha de destacar que desde la versión 2.1.33 de Linux el módulo `parport`, creado a partir de este proyecto, se encuentra fusionado con la distribución principal gestionada por Linus Torvals.

Simultaneamente a la creación del parport se diseñó el EPLIP que proporciona una interconexión sobre puerto paralelo soportando los nuevos modos definidos por el estándar IEEE1284.

El nuevo módulo EPLIP aumenta el ancho de banda un 400 % y reduce las latencias un 50 % con respecto a las versiones anteriores que existían en Linux.

1.3. Organización de la documentación

La documentación se encuentra estructurada en cuatro partes diferentes.

Introducción: La primera de las partes, y solo quiere dar una visión global del alcance, objetivos y organización del proyecto.

Conocimientos básicos: Trata los aspectos básicos que se deben conocer profundamente, para poder iniciar el proyecto con una buena base y suficiente visión para ello. En esta parte hay cuatro focos principales de estudio. El primero trata de las redes TCP/IP, desde el punto de vista estructural, de protocolo y conceptual. El segundo de los temas, es el sistema operativo Linux. El tercer tema explica cuales son las principales características del puerto paralelo. El último de los temas, da una breve explicación de algunos de estos sistemas que se ha creído que pueden ser de mayor importancia tanto para tener un nivel de comparación entre sí, como por la buena proyección futura de alguno de ellos.

[1] Es un proceso de continua evolución que todavía continua

Implementación: Expone los puntos referentes al diseño y la implementación del sistema al completo, así como las distintas evoluciones y alternativas que se han ido manejando a lo largo de su historia. Los diferentes temas en esta parte, se focalizan en primer lugar en una explicación conceptual del diseño global, para pasar a explicar cada uno de los módulos que se compone el sistema, empezando por los de nivel más bajo. Para finalizar, la parte de implementación, hay un capítulo dedicado a las conclusiones deducidas de la experiencia adquirida a través de la implementación de este proyecto. También hay una parte de resultados preliminares que permiten determinar la bondad de la aplicación del proyecto en entornos de desarrollo y comerciales. Finalmente una valoración y conclusiones finales.

Apéndices: Incorporan la bibliografía, así como la licencia de GNU sobre la que se ha desarrollado todo el proyecto.

Parte II.

Conocimientos Básicos

Capítulo 2 ❑

Las redes TCP/IP

En este capítulo se describen las principales características de la *suite* TCP/IP, incorpora una descripción del funcionamiento. Los estándares que forman el conjunto TCP/IP especifican unos protocolos organizados en diferentes capas y módulos que interaccionan entre si, de manera que una red es del tipo TCP/IP si utiliza varios de estos módulos para su interconexión.

2.1. Introducción

La necesidad de comunicación es tan antigua como la propia humanidad, desde que se crearon los ordenadores surgió la necesidad de compartir información. En el momento que se interconectaron los primeros ordenadores nació la primera red. Desde estos comienzos las cosas han cambiado considerablemente, existen protocolos de comunicación entre diferentes máquinas. Una forma de comunicarse entre ordenadores lo consiste las redes TCP/IP.

Las redes TCP/IP se crearon en el departamento de defensa de los Estados Unidos, DARPA, fue un diseño de finales de los años sesenta para estandarizar la red interna de defensa, aunque no entro en funcionamiento hasta 1975. Interesados en que toda la industria armamentística utilizara este sistema de interconexión aparecieron, a principios de los ochenta, en los estándares en las ahora conocidas RFC [J.P80] [J.P81] [Plu82].

En los últimos años la tecnología de redes ha experimentado un crecimiento extraordinario, provocado por diversos factores como pueden ser el paradigma cliente-servidor, la necesidad cada vez mayor de compartición de información, la evolución técnica de los dispositivos de interconexión, pero el gran boom comercial ha sido provocado por la aparición de la red mundial denominada *Internet*.

Por todos estos factores y otros que no se analizan en esta memoria, las redes han llegado a introducirse en nuestras vidas (en la oficina y en casa) de manera natural, y ahora incluso nos puede ser costoso el hecho de imaginar una estructura informática sin considerar la interconexión de red.

Existe una gran variedad de tecnologías y máquinas en el mercado, de esta forma han aparecido gran cantidad de siglas, nombres de redes, protocolos y servicios, pero quizás es fácil, que de entre tantos nombres y siglas, podamos asociar al llamado TCP/IP con

el mundo de Internet y servidores UNIX.

Los estándares que forman el conjunto TCP/IP especifican unos protocolos organizados en diferentes capas y módulos que interaccionan entre si, de manera que una red es del tipo TCP/IP si utiliza varios de estos módulos para su interconexión. De igual manera, se puede deducir es que cualquier máquina que pretenda poder hablar con otra que utilice la *suite* TCP/IP, debe poder utilizar también estos módulos. Por extensión, se puede deducir, que cualquier máquina que desee conectarse a la red *Internet* deberá igualmente implementar la *suite* TCP/IP.

2.1.1. Organización del capítulo

El capítulo que trata las redes TCP/IP tanto en su aspecto funcional como estructural, está organizado en 6 partes diferentes, algunas de las cuales pueden ser ignoradas dependiendo de las capacidades o intereses del lector.

La primera de las partes, trata de la vision global y estructura de la *suite* TCP/IP, viendo su organización por niveles y repasando sus conceptos más fundamentales.

El capítulo continua analizando más a fondo el nivel inferior, llamado nivel de enlace, seguido de un repaso a uno de los módulos relacionados como es el protocolo ARP. Una vez vistos estos conceptos, se continua con la explicación del nivel de red, y sus características básicas de direccionamiento, el concepto de *routing*...

A continuación de explica el siguiente nivel, llamado nivel de transporte, remarcando dos pilares como son el protocolo UDP y TCP [1]. El capítulo finaliza con una breve explicación de los protocolos más comunes que se implementan en en nivel de aplicación.

[1] Existen más protocolos sobre IP como el T-TCP, pero no se encuentran tan ampliamente distribuidos como el UDP y TCP

2.2. Estructura del protocolo

Originalmente la interconexión entre diferentes sistemas no poseía una estructura clara, esta falta de modularidad dificultaba el desarrollo y el mantenimiento. Con la intención de solucionar este problema la ISO creo la conocida torre OSI, *Open System Inteconnection*. El protocolo TCP/IP se creo de forma paralela al estándar OSI por ese motivo existen ciertas diferencias, aunque siguen un modelo de capas.

La estructura en diferentes capas o niveles lógicos, da una mayor modularidad al sistema, permitiendo poder variar alguno de estos componentes, sin necesidad de tocar el resto. Esta característica se hace especialmente importante si tenemos en cuenta que en el mercado existe una gran variedad de ofertas, empezando desde los sistemas operativos (Unix, Windows ™, Mac...), pasando por las aplicaciones (Web, News, Tranferencia de ficheros...) y acabando en los tipos de tarjetas de red (Ethernet, Token-ring, ATM...), y que por tanto, utilizando una estructura de capas se pueden incorporar más rápida y eficientemente nuevos módulos en la estructura global.

El conjunto de protocolos que forman el TCP/IP[2] nació a finales de los sesenta, de un proyecto investigación sobre redes de conmutación de paquetes, financiado por el gobierno de los estados unidos.

TCP/IP, aunque muchas veces se catalogue como protocolo, realmente es una combinación de diferentes protocolos en diferentes capas. Es un sistema abierto en su definición de protocolos y existen varias implementaciones disponibles publicamente. TCP/IP es la base de la red Internet, que se extiende a lo largo del planeta con millones de ordenadores conectados entre si.

2.2.1. Arquitectura por niveles

El conjunto de protocolos del TCP/IP, se considera dividido normalmente en cuatro capas generales diferenciadas, tal y como se observa en la figura 2.1.

Nivel de enlace : Es el nivel que abstrae los protocolos superiores de los detalles *Hardware* del tipo de conexión que se utiliza. El objetivo de este nivel es de establecer, mantener y finalizar la conexión entre dos puntos conectados en un mismo interfaz físico.

Nivel de red : Esta capa es la encargada del movimiento de los paquetes por la red, el *routing* permite que paquetes de diferentes niveles de enlace puedan comunicarse.

Nivel de transporte : Proporciona una comunicación fiable independientemente de la red de interconexión que se utiliza, desde el punto de vista de niveles superiores mantiene un *flujo* de información entre dos máquinas. Hay varios protocolos diferenciados en este nivel, por un lado el TCP (*Transport Control Protocol*) el cual proporciona un sistema de transmisión seguro en cuanto a división, ordenación y reenvío de paquetes. Y por otro lado el UDP (*User Datagram Protocol*) el cual

[2]TCP/IP: Transport Control Protocol /Internet Protocol

proporciona un sistema mas simple que el TCP, consistente en solo encargarse de enviar los paquetes al host destino, sin ningún tipo de reordenación. El T-TCP, es de reciente creación y esta orientado a transacción, eliminando alguna de las deficiencias del TCP.

Nivel de aplicación : Es el encargado de los detalles de la aplicación, el cual puede estar formado por distintos subniveles.

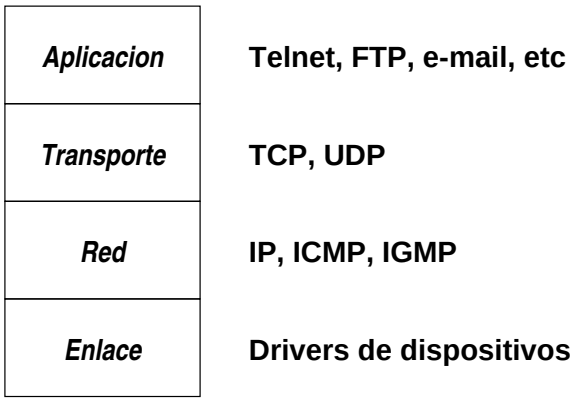

Figura 2.1.: Capas del TCP/IP

Mientras que TCP/IP se puede distinguir en 4 capas principales a la que se tendría que añadir en nivel físico y el nivel de usuario, la torre OSI se divide en 7 niveles. La figura 2.2 muestra la relación entre OSI y TCP/IP, se puede observar que el nivel de enlace de TCP/IP es ligeramente más potente desplazando el IP y TCP. Una explicación más extensa del estándar OSI así como una comparación con TCP/IP se puede encontrar en [S.T91].

2.2.2. Encapsulamiento

La forma de poder utilizar e intercambiar información de los diferentes protocolos entre diferentes máquinas, es la del *encapsulamiento*. Esta técnica consiste en transmitir paquetes de un nivel, hacia el mismo nivel de la máquina remota [3]. Esto se consigue *encapsulando* los diferentes tipos de datos de un protocolo dentro de los datos del protocolo inferior, así sucesivamente hasta el nivel inferior.

Los datos del nivel inferior son físicamente transferidos a la máquina destino, y entonces cada protocolo se encarga de desencapsular los datos dentro de su propio "paquete" y pasar-los a su nivel superior. De esta manera, en la zona de *datos* de un "paquete" de un protocolo, hay un "paquete" de un nivel superior.

[3] Aunque se hable siempre de transmisión entre diferentes máquinas, se debe tener en cuenta, que los sistemas operativos actuales, permiten la utilización del TCP/IP dentro de la misma máquina, simulando toda la torre de protocolos, por la cual cosa se puede utilizar cualquier servicio TCP/IP con la misma máquina destino y origen.

<table>
<tr><th></th><th>OSI</th><th>TCP/IP</th></tr>
<tr><td>7</td><td>Aplicacion</td><td>Usuario</td></tr>
<tr><td>6</td><td>Presentacion</td><td>Telnet,FTP ...</td></tr>
<tr><td>5</td><td>Sesion</td><td>(no existe)</td></tr>
<tr><td>4</td><td>Transporte</td><td>TCP o UDP</td></tr>
<tr><td>3</td><td>Red</td><td rowspan="2">IP</td></tr>
<tr><td>2</td><td>Enlace</td><td>Enlace</td></tr>
<tr><td>1</td><td>Fisico</td><td>Fisico</td></tr>
</table>

Figura 2.2.: Correspondencia entre TCP/IP y OSI

La manera de poder saber que tipo de datos contiene cada "paquete" de un protocolo, es utilizando una estructura de *cabecera+datos* para cada una de los bloques de información a enviar o recibir.

La cabecera es una definición de la estructura de datos, dependiente de cada nivel y protocolo. Por ejemplo, la definición de la cabecera y datos de una trama [4] Ethernet del tipo 802.2 se puede observar en la figura 2.4.

Donde en la cabecera se especifica:

- La dirección *Ethernet* origen y destino de las tarjetas que intervienen en la comunicación.
- El identificador del protocolo al que corresponde el tipo de datos que transporta la trama. Este valor nos indica si los datos que transporta la trama pertenecen a un paquete IP, IPX, ARP... y se usa en la recepción (desencapsulamiento) para entregar-lo al protocolo superior indicado.

Esta división por capas permite una fuerte abstracción de los niveles inferiores, produciendo así el efecto de que cada nivel de protocolo, solo debe saber como "dialogar" con paquetes de su mismo nivel.

[4] Los conceptos de trama o paquete poseen un mismo significado, aunque trama se usa preferentemente en los niveles inferiores mientras que paquete se usa en los niveles superiores

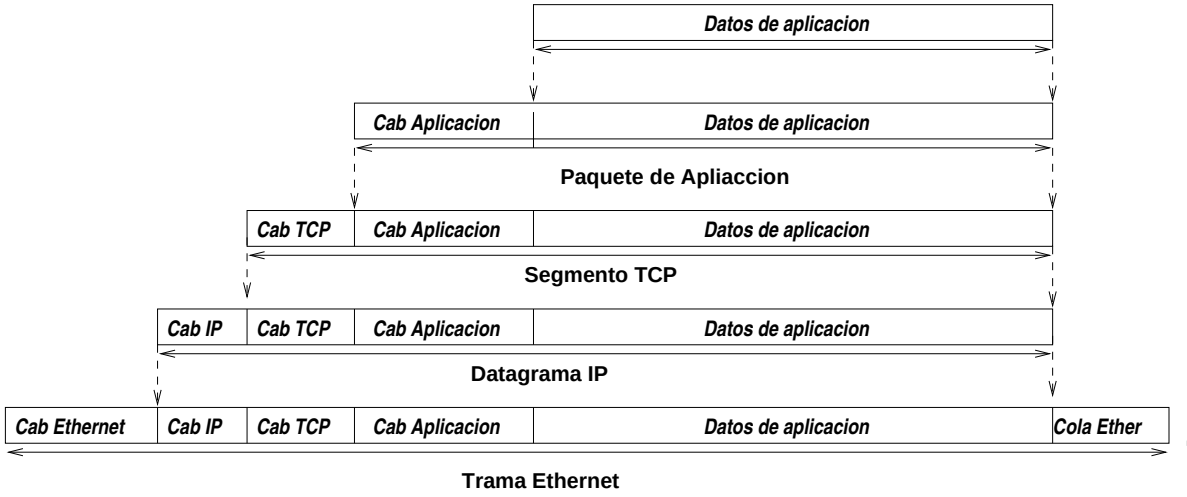

Figura 2.3.: Sistema de encapsulamiento por niveles

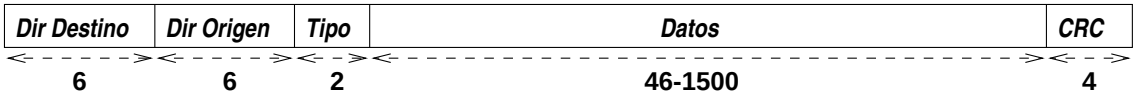

Figura 2.4.: Trama Ethernet

2.2.3. Ejemplo ilustrativo

Cuando un usuario esta realizando una sesión *ftp*, File Transfer Protocol, contra otra máquina, cada paquete de información entre las dos, deberá pasar por los siguientes pasos:

1. La aplicación preparará un paquete de su tipo, con los datos (por ejemplo) del fichero que el usuario quiere transferir, y lo pasará al modulo de TCP del nivel inferior.

2. El modulo de TCP preparará un paquete de su tipo, rellenará su cabecera con los datos apropiados y copiará el paquete que le ha pasado el nivel superior, íntegramente (con cabecera y datos) en la zona de datos de su paquete. Cabe notar, que en la cabecera se deberá indicar que el tipo de datos que transporta el paquete es del tipo "ftp". Finalmente se pasará el paquete al modulo IP del nivel inferior.

3. El modulo IP, volverá a hacer lo mismo, que el anterior, es decir, encapsular el paquete pasado dentro de un nuevo paquete IP.

4. El modulo de enlace volverá a repetir la operación con el paquete procedente de modulo IP, creando una trama adecuada para el tipo de transporte que implemente.

Esta trama serà transferida por la red hacia la maquina destino[5], y recibida por el nivel de enlace de esta.

5. EL nivel de enlace extraerá el paquete encapsulado en la parte de datos de su trama, y viendo que es del tipo IP (indicado en la cabecera) lo pasará al módulo IP para que lo trate.

6. El Módulo IP también desencapsulará el paquete de su zona de datos y lo pasará al TCP(indicado en su cabecera)[6].

7. El módulo TCP hará la misma operación de desencapsulamiento y finalmente lo pasará al módulo superior de "ftp".

8. En este momento el programa de "ftp" ya ha conseguido pasar unos cuantos datos de una máquina a otra sin importarle que tipo de transporte, ni de red tenga por debajo.

El nivel lógico que se emplea en este ejemplo, tiene la disposición que muestra el siguiente diagrama de la figura 2.5.

Como se ha visto, desde el punto de vista del nivel de "ftp", los datos que debe conocer, solo son los del tipo "ftp", ya que así es como los transmite, y así es también como los recibe. Igualmente pasa con los otros niveles, por tanto este tipo de distribución permite una abstracción importante entre cada uno de ellos.

2.2.4. Conceptos de routing

Cuando dos máquinas quieren intercambiar información, estas se pueden encontrarse ubicadas en diferentes entornos de red. Según el tipo de topologia de red donde se encuentren, el comportamiento de los niveles inferiores varía sensiblemente:

- Si las máquinas se encuentran físicamente situadas en el mismo segmento de red (mismo cable físico [7].) las direcciones de la cabezera del nivel de IP serán las de las propias máquinas origen y destino que intervienen en la transacción. Las direcciones de la cabezera de nivel de enlace serán las propias direcciones de enlace de las máquinas que intervienen. En el caso de tarjetas Ethernet, sus direcciones Ethernet.

- Si las máquinas se encuentran en redes distintas (la conexión se debe de realizar a través de una o más máquinas intermedias) las cabeceras IP seguirán siendo las mismas que en el primer de los casos, pero las dirección destino de enlace, pasará de

[5]En el ejemplo, no se tiene en cuenta la resolución de direcciones de nivel de red, ver sección de ARP para más detalles

[6]Dependiendo de la configuración de la red, en caso de haber máquinas router entre medio de las máquinas origen i destino, el nivel IP, podría retransmitir otra vez el paquete entre máquinas intermedias

[7]El caso de que haya un *bridge* entre medio no se contempla como un caso separado

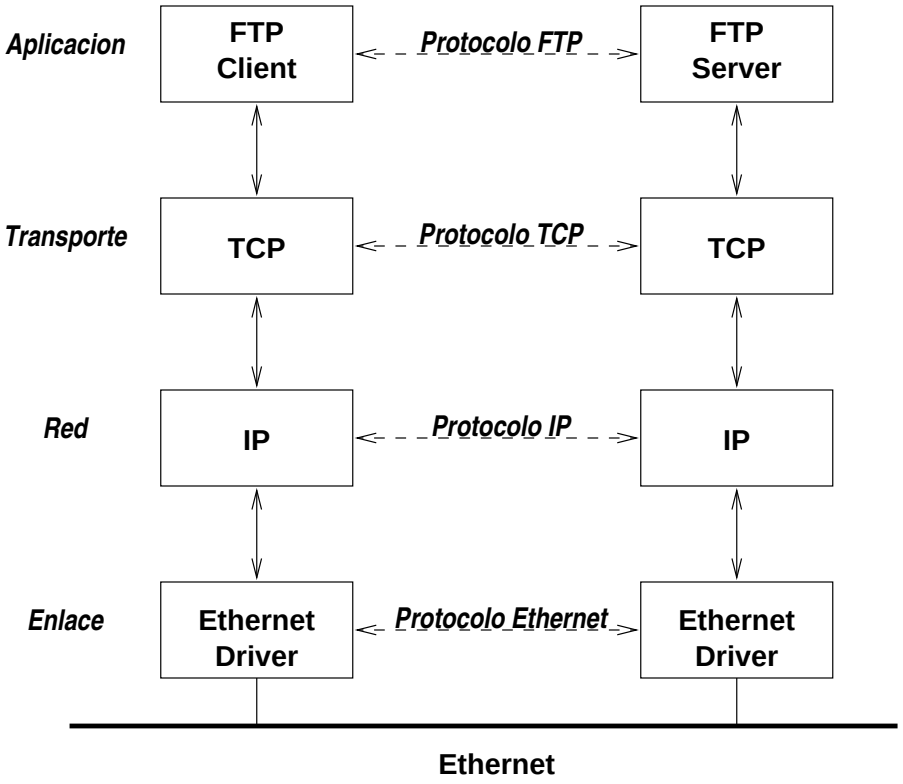

Figura 2.5.: Encapsulación usada en la transmisión FTP

ser la dirección de enlace de la próxima máquina intermedia en el camino hacia la máquina destino. Así pues, en cada uno de los "saltos"(comúnmente denominados *hops*) la dirección de enlace destino ira cambiando, para ser la de la próxima máquina intermedia, mientras la dirección IP destino no variará.

Teniendo en cuenta que la *suite* TCP/IP se comporta, básicamente, tal y como se ha explicado en esta sección, debe notarse que hay diferentes variaciones o ampliaciones en algunos de los protocolos que algunos sistemas operativos han ido introduciendo para aumentar las funcionalidades y mejorar algunos aspectos. Algunas de estos casos pueden ser el IP-aliasing, IP-masquerading, Proxys(ARP, IP)...

No es el motivo de este capítulo el dar abasto a estos temas por tanto una mayor explicación se puede encontrar en [Ste95]

2.3. El nivel de enlace

El nivel de enlace es el nivel más bajo de la jerarquía de protocolos del TCP/IP. Este nivel es el encargado de todo lo referente a la parte *hardware* de la comunicación, mostrando una interfaz homogénea hacia el nivel de IP.

El protocolo TCP/IP, tal y como ya se ha dicho, por su estructura modular, puede soportar distintos tipos de interfaces hardware subyacentes. De esta manera es posible utilizar diferentes tecnologías para la comunicación física, ya sea entre máquinas con diferente tipo de conexión, así como en una misma máquina con mas de una interfaz de red de diferente tipo.

Algunas de las interfaces más comunes que podemos encontrar son:

- Ethernet
- Token ring/Bus
- FDDI (Fiber Distributed Data Interface)
- Puertos serie tipo RS232

Cada uno de estas interfaces, suele definir una estructura de cabecera (o más de una[8]), la cual contenga como mínimo, los datos necesarios para poder realizar transmisiones de información. Lo más habitual suele ser, definir los siguientes paràmetros:

- Dirección Origen
- Dirección Destino
- Tipo de paquete encapsulado

Por ejemplo, estos son los datos que definen una cabecera Ethernet. Seguidos de estos datos de cabecera se encuentran los datos, y al finalizar los datos, en el caso de Ethernet, se define otro campo (*trailer*) para la detección de errores.

El aspecto de una trama Ethernet es el que muestra la figura 2.6.

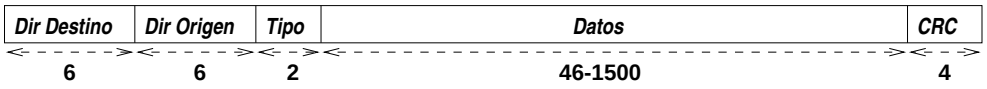

Figura 2.6.: Aspecto de una cabecera Ethernet

Las direcciones origen y destino del nivel de enlace, corresponden a direcciones asociadas al dispositivo físico que transmite la información. Estas, según el tipo de dispositivo

[8]En el caso de las tarjetas de protocolo CSMA/CD, hay dos tipos de encapsulamientos definidos, como el IEEE 802.2/802.3 [JR88] i el Ethernet [Hom84] para que además puedan compartir el mismo medio.

o de medio, son opcionales. Por ejemplo, si el medio no es compartido, es decir que se trata de un medio donde solo hay dos posibles máquinas para la comunicación en el mismo instante, el sentido de las direcciones origen y destino de nivel de enlace, puede dejar de tener sentido. Esto es lo que ocurre en una comunicación punto a punto *point-to-point*, como por ejemplo utilizando un puerto serie entre dos máquinas. En el caso de los dispositivos como por ejemplo Ethernet, las direcciones son formadas por 6 octetos identificadores unívocos de cada una de las tarjetas de que se compone la red.

El campo de tipo en la cabecera es especialmente importante, debido a que nos indica el tipo de trama que está encapsulada en la zona de datos de la trama. Este campo también puede ser no necesario, si por ejemplo solo utilizamos un mismo protocolo superior con el mismo tipo de dispositivo. Por ejemplo, en el caso de que siempre fueramos a utilizar el IP como protocolo de nivel de red, en una conexión de puertos serie, no haría falta un campo de tipo. El posible problema, podría venir si fuera necesario utilizar ese medio para transmitir IPX[9] sobre y TCP/IP simultaneamente.

La cabezera de pueden añadir tantos campos como sea necesario, dependiendo de las funcionalidades que queramos que posea el tipo de dispositivo. Por ejemplo en el protocolo PPP (Point to Point Protocol) hay un campo de protocolo, que indica si la zona de datos corresponde a un paquete IP, o corresponde a información de control de enlace (LCP Link Control Protocol [Sim94] o de red (NCP Network Control Protocol [McG92]. De esta manera multiplexa paquetes de información de la conexión, con paquetes de información de datos IP.

En el nivel de enlace existe un concepto conocido como MTU (*Maximum Transfer Unit*) que se asocia a cada dispositivo, y define el tamaño máximo que puede llegar a tener un paquete/trama sobre ese medio. La MTU es dependiente del tipo de medio a transmitir, por ejemplo en medios que necesiten un tiempo de *setup* de red alto (por ejemplo negociación de medio) se tenderá a tener valores más altos.En medios con latencias muy bajas con aplicaciones de orientadas a imágenes o voz, se tenderá a MTUs menores para no apreciar retardos significativos.

En la tabla 2.1 se pueden observar los valores más apropiados para varios tipos de enlaces físicos.

Cabe notar que el tamaño de la MTU entre máquinas en redes distintas es importante para evitar fragmentación de paquetes. Por tanto hay mecanismos definidos como el algoritmo *Path MTU discovery* [JM90]. En [Ste95] se explica como encontrar la mínima MTU de transmisión entre dos máquinas en distintos tipos de redes.

2.4. El protocolo ARP/RARP

Dentro de la misma red física pueden existir diferentes tipos de protocolos mezclados, utilizando los mismos medios a nivel de enlace. El modulo de enlace, trabaja directamente con las direcciones de su tipo hardware, por lo tanto, debe de haber un tipo de "protocolo" o modulo que se encargue de poder traducir o encontrar que direcciones

[9]IPX es la definición del nivel de enlace que utiliza Novell en sus redes.

Red	MTU(Bytes)
Hyperchannel	65535
16Mbits/s Token Ring (IBM)	17914
4Mbits/s Token Ring (IEEE 802.5)	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576
Point-to-Point (baja espera)	296

Cuadro 2.1.: Ejemplos de MTU

de enlace corresponden las direcciones que vienen del nivel superior. El modulo que se encarga de hacer este "mapeo" de direcciones de entre el nivel de enlace y el nivel de red es el modulo de ARP/RARP que se encuentra situado lógicamente entre ambos niveles.

Las siglas ARP derivan de *address resolution protocol*, es decir, protocolo de resolución de direcciones. De la misma manera, RARP *Reverse Address Resolution Protocol* significa protocolo de resolución inversa de direcciones, que se usa de forma un poco distinta, básicamente orientado a máquinas sin disco propio y terminales-X.

El protocolo ARP funciona en modo de pregunta respuesta. Cuando una máquina quiere enviar datos a una cierta dirección de red , ésta debe saber a que dirección de enlace corresponde. Para averiguar esta dirección, la máquina preparará y enviarà un paquete del tipo ARP (*query*, donde se especificaran las siguientes direcciones:

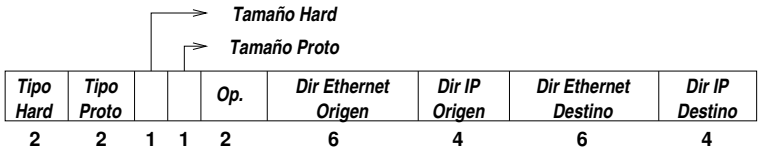

Figura 2.7.: Aspecto de trama ARP

- La dirección origen de nivel de red. Será la dirección IP, IPX... de la máquina que busca resolver la dirección.
- La dirección destino de nivel de red. Será la dirección IP, IPX... de la máquina de la cual se quiere saber su dirección de enlace.
- La dirección origen de nivel de red. Será la dirección Ethernet, Token ring... de la máquina que inicia la pregunta.
- La dirección destino de nivel de red. Es precisamente la dirección que queremos saber. En el paquete se especificará la dirección *broadcast* del nivel de red. Por ejemplo en Ethernet, serán los 48 bits a '1'(FF:FF:FF:FF:FF:FF).

El significado genérico de una dirección *broadcast* en una red, significa que los destinatarios del paquete, son todas aquellas máquinas físicamente conectadas a la misma red. Por tanto, todas aquellas máquinas que reciban un paquete de *broadcast* deben procesarlo como si fuera destinado a ella.

Esta afirmación no es del todo cierta, ya que pueden existir máquinas que hagan de *proxy* de direcciones ARP, actuando así de intermediarios entre máquinas en redes físicamente distintas.

Una vez que una máquina recibe un paquete ARP *query*, y comprueba que la dirección de nivel de red destino es la suya, prepara un paquete respuesta ARP *reply* con la misma información [10], pero esta vez también incluyendo su dirección de nivel de enlace.

Una vez que la máquina originaria del *query ARP* recibe el *reply*, ya puede saber la dirección de nivel de enlace que buscaba, porque en el paquete recibido incorpora toda toda la información necesaria.

Tal como se ve, es un proceso de pregunta-respuesta bastante costoso, ya que para enviar un paquete de datos, hace falta enviar 2 paquetes de ARP además del paquete de datos. Para evitar este exceso de reenvío el modulo de ARP trabaja con unas caches de los *mappings* de direcciones, la cuales permiten no crear los paquetes de ARP en caso que las direcciones se encuentren ya el las tablas de la cache. Cada una de las entradas a las tablas, tiene asociado un *time-out* y por tanto cada un cierto tiempo, caducan y se debe volver a proceder con el protocolo anteriormente explicado.

El protocolo ARP genéricamente debe traducir direcciones de cualquier nivel superior (red) a direcciones de cualquier nivel inferior (enlace). Sin embargo, la mayoría de los módulos implementados hasta hace poco tiempo, solo permitía la traducción de direcciones IP a Ethernet. A medida que nuevas tecnologías han ido apareciendo, algunos de los sistemas operativos mas importantes han ido adaptándose a ellos. Así por ejemplo, mientras que *BSD4.4* sólo permite *mappings* del tipo Ethernet-IP [MBJS96], el sistema operativo *Linux* permite la traducción de direcciones (Ethernet, FDDI, MIRYNET) en direcciones (IPv4, IPv6...).

El protocolo RARP, es del mismo estilo, y permite a las máquinas, a partir de su dirección de enlace, obtener una dirección de red. Por ejemplo permite a los terminales-X, al botar, conocer que dirección IP les corresponde, y poder empezar ha enviar y recibir paquetes por la red. Este sistema requiere una configuración manual por parte del administrador, el cual debe haber configurado una máquina para que atienda las peticiones de las direcciones de enlace apropiadas.

[10] Se debe notar, que los roles de destino y origen cambian porque ahora la máquina preguntada es la emisora del paquete, por lo tanto los campos origen y destino se refieren ahora a la máquina contraria del paquete ARP *request*

2.5. El nivel red

El nivel de red *Internet Protocol* es quizás uno de los módulos más importantes del conjunto de protocolos TCP/IP, ya que todos los paquetes de los niveles superiores (TCP, UDP...), llegado a un punto, se convierten en paquetes *IP* para ser transmitidos hacia su destino.

A parte del módulo de IP. existen otros protocolos, como el ICMP (*Internet Control Messaging Protocol*) o el IGMP (*Internet Group Management protocol*) que estarían un poco por encima del nivel de IP, pero pero no los tendremos demasiado en cuenta, debido a que las funcionalidades que ofrecen son muy distintas de las que ofrece el módulo *IP*

El protocolo *IP* provee a los protocolos superiores, un servicio de conexión con dos características fundamentales:

- Un sistema de transmisión **no fiable**, ya que no garantiza en ningún momento que los paquetes enviados lleguen a su destino.

- Un sistema **no orientado a conexión**, la cual cosa significa que no se guarda ninguna información ni relación entre paquetes sucesivos.

La manera de tratar el hecho que no haya control de errores (por el supuesto de que la transmisión no fiable), se intenta suplir con una política de hacer lo que mejor se pueda hacer en cada momento. Por ejemplo, si un paquete no puede ser transmitido por un router, por cualquier circunstancia del momento, éste, perderà el paquete, i tratará de enviar una notificación al originario para que éste pueda saber lo que ha pasado. Éste simple protocolo de control de errores, y notificación de mensajes se llama ICMP (*Internet Control Message Protocol*).

Por tanto, si un protocolo o nivel superior, necesita hacer uso de un sistema de transmisión fiable, o mantener una relación o numeración de los paquetes a enviar, lo debe implementar el mismo protocolo, o utilizar otro de nivel intermedio.

Una de las funcionalidades más importantes del protocolo IP, es el enrutamiento de paquetes. Las direcciones con las que trabaja este nivel son las conocidas direcciones IP, que constan de cuatro octetos en el caso de la especificación de versión 4 (*IPv4*). Esta especificación es la más utilizada y extendida por todo el mundo, sobretodo por la gran extensión que ha llegado a tomar *Internet*, siendo *IPv4* su base de comunicación.

Debido, precisamente a su gran extensión y utilización, la definición inicial se ha empezado a quedar pequeña, sobretodo porque tampoco estaba pensada para llegar a ser una red tan grande como ha llegado a ser. Por este caso, ya hay una nueva definición del protocolo *IP* pensada para abarcar a un numero de máquinas mucho más grande, encarada para grandes redes. Ésta definición es conocida como *IPv6* de la que más adelante se darán más detalles.

2.5.1. Los paquetes IP

Los paquetes del protocolo *IP* [11] tienen el siguiente aspecto, y los siguientes datos en su cabecera:

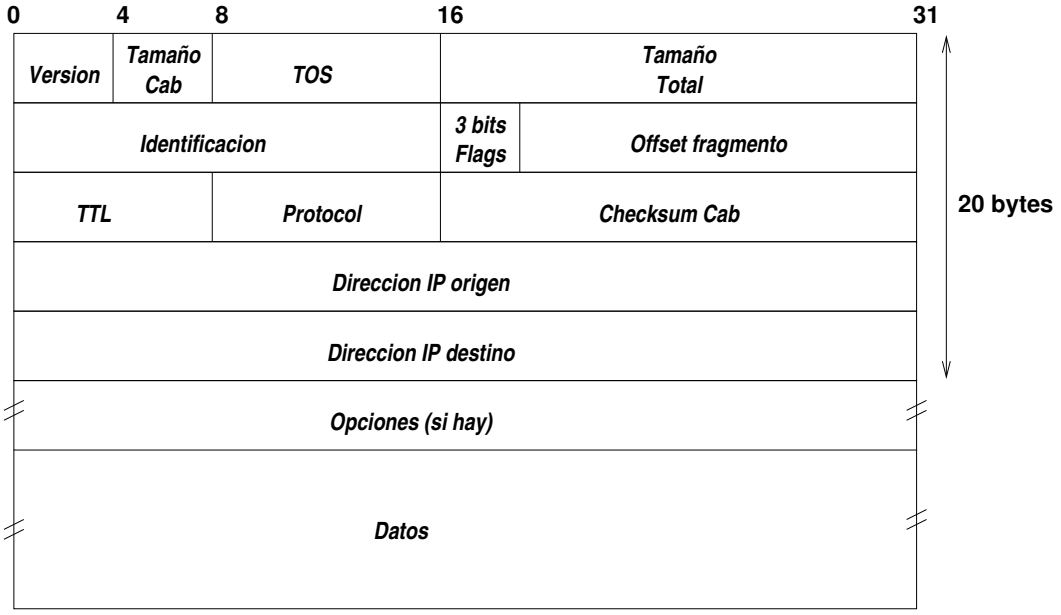

Figura 2.8.: Aspecto de un paquete IP

Los campos más importantes a tener en cuenta son:

- Versión del protocolo (*IPv4-IPv6* por el momento).
- Longitud de la cabecera (que puede ser variable) y longitud de la zona de datos.
- TTL (*Time To Live*) indica el numero máximo de máquinas intermedias (routers) por las que puede pasar el paquete.
- Un *checksum* de validación de la cabecera.
- el protocolo que viene encapsulado en la zona de datos
- Las direcciones origen y destino de nivel de red.

Debido a que diferentes tipos de máquinas intercambian información, y usan diferentes tipos de interpretación a nivel de bits (máquinas *little-endian* o *big-endian*) se define que el tipo de transmisión (y interpretación) para los datos TCP/IP que se transmitan por la red, debe ser del tipo *big endian*. Esta representación se la denomina *network byte*

[11] Cuando se hable del protocolo *IP* sin ninguna otro matiz, se considerará que se habla de la definición *IPv4*, en otro caso, para dar referencia a la versión extendida, se hará en forma explícita

ordering (ordenamiento de los bytes de la red), consiste en que el orden de transmisión de los bits, en cada bloque de 4 bytes, se haga de forma que los primeros sean los bits 0-7, seguidos por los bits 8-15, 16-23 y finalmente 24-31. De esta manera cada tipo de máquina puede interpretar correctamente los datos que recibe, independientemente del tipo de representación que use.

2.5.2. Máscaras y direcciones

Cada dirección IP, tal y como ya se ha dicho, consta de un grupo de cuatro bytes, donde pueden darse cifras de rango 0-255 en cada uno de ellos.

Por ejemplo la dirección de una de las máquinas servidoras de la universidad es **130.206.42.222** en notación decimal [12].

Las diferentes máquinas que utilizan el TCP/IP, están agrupadas de manera virtual o física, en distintas redes, la cual cosa permite un cierto orden en su asignación, y además permite una distribución jerárquica.

Esto se consigue, interpretando, parte de la dirección IP como un número de red, y parte como numero de la máquina dentro de esta red. El limite de esta división viene marcado por una máscara de red, que va siempre asociada a una dirección IP.

Dependiendo del tamaño de los primeros bits, las direcciones IP se clasifican por clases. Tal y como muestra la tabla 2.2 la dirección IP se catalogan en clases según si su dirección IP [13].

Clase	**Rango**
A	**0**.0.0.0 hasta **127.255.255.255**
B	**128**.0.0.0 hasta **191.255.255.255**
C	**192**.0.0.0 hasta **223.255.255.255**
D	**224**.0.0.0 hasta **239.255.255.255**
E	**240**.0.0.0 hasta **247.255.255.255**

Cuadro 2.2.: Clases IP

Las diferentes clases, a parte de proporcionar una división de rangos de direcciones, también tienen la característica de estar formadas por diferentes longitudes de bits definidores de red o máquina. Así pues, en una clase A hay 7 bits que definen el numero de red y 24 que definen el número de máquina dentro de esta red. Por tanto cabe notar que son redes con muchas máquinas. Las clases B tienen 14 bits de identificador de red, y 16 de máquina. Las clases C tienen 21 bits de identificador de red y 8 de máquina.

El caso de las clases E, son las direcciones multicast. No se entrará en detalles, pero a "grosso modo" se puede apuntar que son direcciones que se usan para poder llegar a diferentes grupos de máquinas a la vez. De alguna manera podría compararse a hace

[12] Normalmente, la manera de representar una dirección IP, es en forma decimal y separando las cifras por puntos

[13] Las direcciones IP desde 248.0.0.0 hasta 255.255.255.255 están reservadas para otros usos

un *broadcast* a un grupo selecto de máquinas, no necesariamente en la misma red física. Para más información consultar capítulo 12.4 de [Ste95].

A continuación se explicará un caso práctico con la descripción de las direcciones IP que posee asignadas la Universidad Ramon Llull [14].

La dirección de la red asignada a la universidad es 130.206.42.0. Si observamos la tabla anterior, podemos ver que corresponde a una clase B aunque la Salle solo poseía una delegación de tipo clase C. por tanto puede contener un total de 256[15]máquinas en ella (8bits).

Por tanto, en el caso del servidor con dirección 130.206.42.222, tendremos que interpretarlo como: Dirección de red 130.206.42.0 y dirección de máquina 222 dentro de esa red. Así mismo, la mascara de red que corresponde a esta división es 255.255.255.0 [16], lo cual define que los 24 primeros bits de la dirección se deben interpretar como dirección de red, y los últimos ocho bits como numero de máquina.

En el caso de las clases A o B, normalmente se usa una técnica denominada *subnetting* que permite subdividir una misma red, en sub-redes más pequeñas. Esto se consigue aplicando máscaras de sub-red más restrictivas que las normales.

2.5.3. El enrutamiento

Una de las funcionalidades más importantes del nivel de IP, es la capacidad de saber enrutar (enviar al destino apropiado) los paquetes que le llegan. El concepto es diferente si estamos hablando de una máquina (*host*) o de un router.

En el caso de un *host* es muy sencillo:

- Si el paquete a enviar es para alguna máquina directamente conectada al host (SLIP, PPP...) o a la red (Ethernet, Token-Ring...) enviárselo directamente.
- En caso contrario enviarlo al router por defecto que tenga configurado, ya que será este quien se encargue.

El caso de un router, es un poco más complicado, porque la tabla de decisiones para el enrutamiento *routing table*, puede ser más compleja. Esta tabla de rutas es la que usará el router para decidir que hacer con el paquete. Cada una de las entradas a esta tabla son posibles rutas a elegir, y contienen la siguiente información básica:

- La dirección IP destino, ya sea un host o una red.
- La dirección IP de la máquina siguiente máquina intermedia en el camino hacia la dirección destino.
- Interfaz por el cual debemos mandar el paquete.

[14]Configuración existente entre 1995 y 1997

[15]Realmente, las maquinas con los todos los bits a cero, o todos a 1 suelen usarse para las direcciones de red genérica y *broadcast* respectivamente.

[16]También existe una codificación altamente extendida que se expresa según el número de bits fijos, en este caso seria 24

El algoritmo de rutas, de manera esquemática, compara la dirección IP destino con cada una de las entradas en la tabla, y lo envía por la que tenga la coincidencia más restrictiva. En caso que de que no se produzcan coincidencias, lo enviará por su ruta por defecto. En caso que tampoco tenga definida una ruta por defecto, podrá generar un paquete del tipo ICMP, para comunicar al emisor del paquete perdido, que no es posible encontrar una ruta apropiada para él.

2.5.4. IPv6

El protocolo IP más conocido y utilizado actualmente se basa en su versión 4. Esta versión presenta algunos puntos "flojos"los cuales se han intentado resolver mediante la nueva versión del protocolo llamada IP *new generation* (IPng) o también IP versión 6 (IPv6).

Los cambios que se han introducido en esta nueva definición del protocolo, se pueden resumir en los siguientes puntos:

Expansión de las propiedades de direccionamiento La IPv6 incrementa el tamaño de las direcciones IP, de 32 a 128, para así poder soportar más niveles en la jerarquía de direccionamiento, un mayor numero de nodos posibles, y una manera más simple de configuración de las direcciones. La escalabilidad de el *multicast routing* se ha también mejorado, añadiendo un campo de *scope* en las direcciones *multicast*. Se ha añadido un nuevo tipo de dirección llamada *anycast address* utilizada para enviar paquetes a alguno nodo de un grupo de ellos.

Simplificación de los formatos de cabeceras Algunos de las cabeceras de la versión IPv4 se han perdido o han sido declarados como opcionales para reducir el procesamiento en los casos más comunes, y para limitar el coste del ancho de banda de la cabecera de IPv6.

Soporte mejorado para extensiones y opciones Han habido cambios en la manera que las opciones son codificadas para mejorar aspectos como el *forwarding*, longitudes menos estrictas o para una mejor introducción de nuevas opciones para el futuro.

Funcionalidad de etiquetamiento del flujo Se introduce una nueva funcionalidad para poder etiquetar paquetes pertenecientes a un cierto tipo de flujo de trafico, por el cual quien lo envía pide un tratamiento especial, como una cualidad diferente de la por defecto, o un servicio de real-time".

Funcionalidades de autentificación y privatización Se definen extensiones de soporte para la autentificación, integridad y (opcionalmente) para la confidencialidad de datos.

El protocolo IPv6, aunque esta definido, aun esta muy poco extendido en el mundo de los sistemas operativos. Para citar un ejemplo, el sistema operativo Linux, que se

caracteriza (entre otras cosas) por ser uno de los más rápidos en la adopción de nuevas tecnologías, ya soporta esta nueva versión, aunque no se pueden encontrar las herramientas para poder utilizarlo completamente, en las distribuciones más comunes de este. Pero en el caso que alguien quiera experimentar, solo tiene que coger las nuevas aplicaciones directamente de los servidores más comunes de Internet.

2.6. El nivel de transporte

El nivel de transporte es donde residen los protocolos que se benefician de las funcionalidades que proporciona el nivel de red. Los protocolos básicos de este nivel son el UDP (User Datagram Protocol) y el TCP (Transport Control Protocol), siendo este último más utilizado por los módulos superiores.

2.6.1. El protocolo UDP

El protocolo UDP, es un protocolo simple orientado a datagrama. Cada operación de salida de un proceso produce un único datagrama UDP, dando así un solo paquete IP a transmitir. Esto difiere significativamente del protocolo TCP, ya que es un protocolo orientado a conexión, y donde los datos del proceso no tienen o no es necesario que tengan relación con el numero de paquetes generados.

Otra de las diferencias básicas entre ambos protocolos, es que mientras el TCP implementa un canal de comunicación fiable, es decir, control de errores, retransmisiones, etc... el UDP sigue sin proporcionarlo. Básicamente por este motivo el TCP es más utilizado, ya que muchos programas tienen necesidad de un control de fiabilidad de la transmisión, y aprovechan así las herramientas que el TCP les proporciona.

La simplicidad del protocolo UDP puede apreciarse en su cabezera, donde solo figuran las direcciones origen y destino del nivel de transporte (llamadas puertos), la longitud de los datos y un campo de checksum para cabezera y datos juntos:

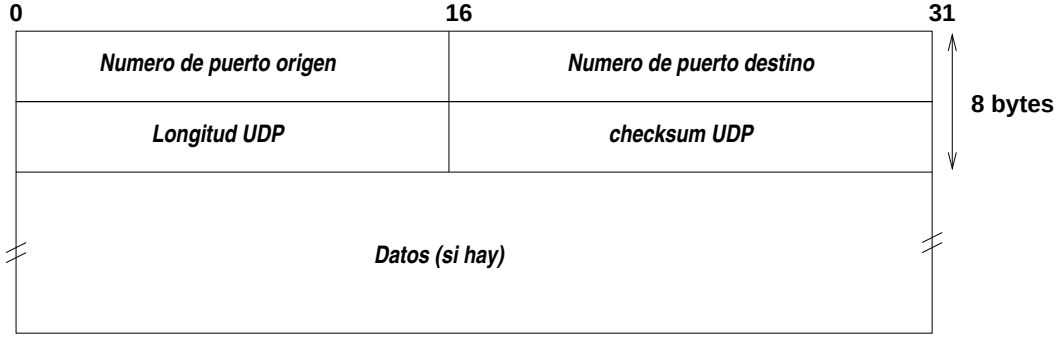

Figura 2.9.: Aspecto de un paquete UDP

2.6.2. El protocolo TCP

Las funcionalidades, que proporciona el TCP son mucho mas complejas que las que proporciona el UDP, y están basadas sobre la seguridad en la transmisión. Los funciones fundamentales que realiza este protocolo, se pueden resumir en los siguientes puntos:

- Los datos de la aplicación son partidos en trozos llamados segmentos.

- Para cada paquete transmitido, se mantiene un contador de tiempo (*timer*) para poder retransmitirlo en caso de no recibir respuesta en el margen indicado.
- Envía un paquete de *acknowledge* (reconocimiento de que el paquete ha llegado) para cada paquete recibido.
- Trabaja con *checksums* en todos los paquetes para detectar posibles errores en la transmisión.
- Gestiona la reordenación de paquetes, ya que el módulo IP, no asegura ningún tipo de orden de llegada de sus paquetes.
- Gestiona los paquetes duplicados.
- Dispone un control de flujo entre máquinas, evitando así acumulamiento de paquetes en máquinas con menos capacidad.

El aspecto de la cabecera, para poder gestionar los puntos anteriores, es el siguiente:

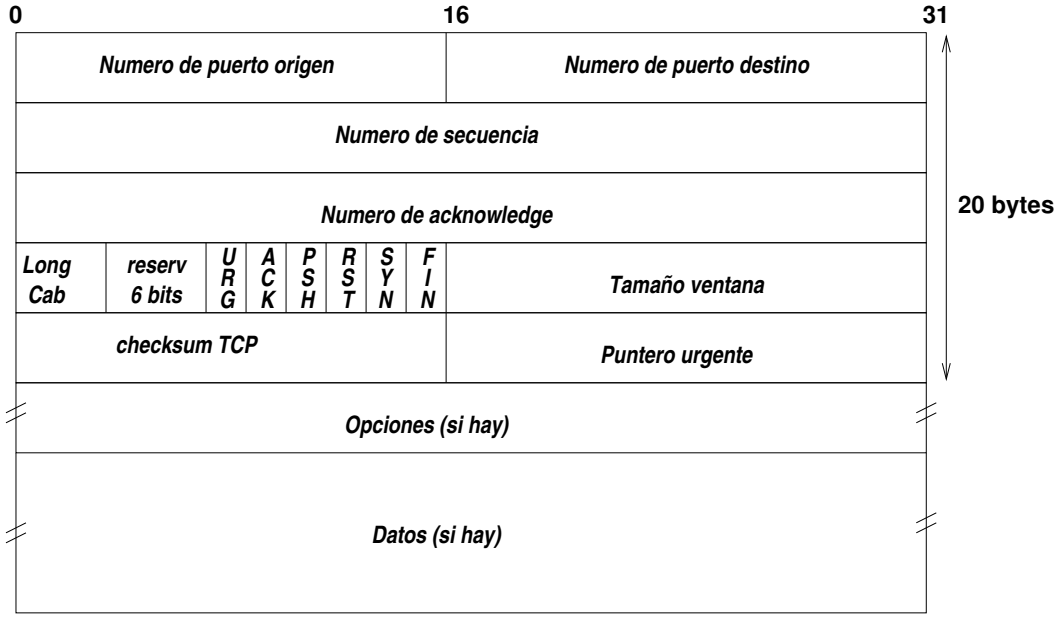

Figura 2.10.: Aspecto de un paquete TCP

El concepto básico del funcionamiento del establecimiento de la conexión en TCP es muy sencillo, y se puede fácilmente comparar a una llamada telefónica, en cuanto a que:

- Hay una persona A, iniciadora de la conversación, quien llama a la persona B con la que quiere intercambiar unas palabras.
- De la manera que sea, se ponen de acuerdo los dos aparatos telefónicos, para crear un canal de transmisión entre ellos, donde las direcciones origen y destino podrían ser los números de teléfono.

- Una vez establecida la conexión, las dos personas ya pueden hablar, hasta que una de las dos decida colgar la conexión.

El caso del TCP es el mismo, ya que primero establece una conexión entre puertos de máquinas, con la salvedad que cuando la conexión ya está establecida, se encarga él mismo de gestionarlo todo para un trasvase de información seguro.

2.7. Nivel de aplicación

Como se ha visto en las secciones anteriores, el conjunto de capas y módulos situados por debajo del nivel de aplicación forman una plataforma, en cierta forma completa, para poder crear aplicaciones de conexión por red. Precisamente esta es la función que ejerce el nivel de aplicación, es decir, crear diferentes y nuevos protocolos de transmisión de más alto nivel, utilizando las funcionalidades que dan el TCP, UDP y los módulos inferiores.

Así pues, daremos un muy breve repaso a algunos de los protocolos con un nivel de aceptación superior, según el protocolo de nivel inferior que utilicen.

Cabe notar que muchos de los protocolos presentados pueden utilizar o utilizan más de un protocolo de nivel inferior. De todas maneras se ha colocado a cada uno de ellos, en el que se considera más significativo.

2.7.1. Aplicaciones basadas en UDP

Estas son las que, generalmente, requieren una menor sincronización y mayor flexibilidad en el tamaño y número de paquetes que deben enviar.

DNS *Domain Name Service* es una base de datos jerárquica y distribuida, organizada por dominios, utilizada para mapear entre nombres de máquinas y sus direcciones en formato IP. A parte de dar información sobre direcciones IP, también se puede utilizar para campos de otro tipo como, para encontrar los servidores de correo de un determinado dominio o listar que máquinas se encargan de gestionar-lo.

TFTP *Trivial File Transfer Protocol* es un protocolo muy sencillo, que es usado con máquinas que carecen de disco (normalmente estaciones de trabajo y terminales X) para que estas puedan "coger"por red, los ficheros necesarios para poder empezar a funcionar. El protocolo es muy simple y funciona sobre UDP (muy simple) para que estas máquinas sin disco lo puedan tener en una pequeña ROM[17].

BootP *Bootstrap Protocol* Es un protocolo que normalmente se una en conjunción con el TFTP, que permite obtener una serie de datos según la dirección de enlace de la maquina que los pide. Es usado por máquinas sin discos, terminales X, impresoras en red..., para conseguir información sobre la configuración IP que deben usar, ficheros que necesitan coger, que después normalmente son recuperados por medio del TFTP. Aunque la función básica se puede comparar al protocolo RARP, lo mejora en dos aspectos básicos:

La información que proporciona consta de bastantes campos más que meramente la dirección IP.

[17]ROM es un término que significa *Read Only Memory* y es un tipo de memoria que se debe grabar con máquinas especiales, y por tanto una vez grabada, no se puede modificar y guarda la información sin necesidad de alimentación eléctrica

Es capaz de redireccionar las peticiones entre redes distintas, evitando así el hecho que se deba tener un servidor por red física.

SNMP *Simple Network Management Protocol* es un protocolo pensado para la gestión de los elementos heterogéneos de las redes TCP/IP. Consta de una parte denominada *manager* quien monitoriza una serie de agentes (*agents*) que son los puntos que se quieren controlar. El manager tiene capacidades de interrogación directa a los agentes, puede también cambiarles parámetros o configuración, y por otro lado, los agentes tienen la capacidad de poder avisar al *manager* de cosas que les haya podido suceder.

NFS/RPC *Network File System* es un protocolo de compartición de ficheros basado sobre el sistema de llamadas a procedimientos remotos de SUN (SUN RPC-*Remote Procedure Call*). NFS es un sistema transparente de cara al usuario, ya que éste percibe el árbol de directorios remoto, exactamente igual que si fuera local.

2.7.2. Aplicaciones basadas en TCP

Los protocolos de aplicaciones que se basan en TCP, generalmente, necesitan una constante sincronización entre máquinas, y utilizan tamaños de paquetes muy fijos.

Telnet & Rlogin Estos dos protocolos proporcionan un sistema de conexión a otra máquina a través de la red, dando la misma funcionalidad que un terminal conectado físicamente en la máquina. Son de los protocolos más populares.

FTP *File Transfer Protocol* es el protocolo por excelencia para la transmisión de ficheros, proporciona una interfaz simple para poder moverse a través de los directorios (locales y remotos), para poder recuperar o copiar ficheros y otras opciones igualmente interesantes.

SMTP *Simple Mail Transfer Protocol* es uno de los protocolos para la transmisión y recepción de correo más utilizado, y permite poder enviar un correo entre dos máquinas a través de TCP/IP.

X es un protocolo con paradigma cliente servidor, que sirve de base a conocido entorno gráfico *X Windows*. Se basa en un servidor (llamado servidor X) el cual acepta conexiones de clientes, que pueden utilizar los recursos que el servidor ofrece, como pueden ser el ratón o parte de la área de la pantalla.

2.7.3. Aplicaciones basadas en otros protocolos

Estas aplicaciones, normalmente son bastante especiales, y normalmente están basadas en funcionalidades administrativas.

Ping es un protocolo que se basa directamente sobre el protocolo ICMP. Es un protocolo muy sencillo, que permite poder enviar y recibir pequeños paquetes en forma de

pregunta respuesta, de diferentes tamaños, para poder testear como se comporta la red entre máquinas origen y destino.

Traceroute es un protocolo mas complejo que el Ping, y que reside directamente encima del protocolo IP. La funcionalidad primordial del protocolo es la de poder crear la lista de máquinas por las que pasa un paquete, para comunicar una cierta máquina origen y destino. Aunque el protocolo Ping tiene algo de estas funcionalidades, el protocolo Traceroute está especialmente diseñado para ello, y resuelve algunos de los problemas que presenta el protocolo Ping, como son la dualidad de caminos de ida y vuelta o otros...

Capítulo 3 ❑

Linux - El sistema operativo

3.1. Introducción

Linux [1] es un sistema operativo de libre distribución compatible con POSIX 1003.1 e incorpora muchas de las funcionalidades de UNIX™ System V y BSD 4.3.

En marzo de 1991 Linus Benedict Torvalds adquirió una licencia del operativo Minix para su i386. Poco más tarde empezó a implementar un nuevo operativo que llamaría Linux. En septiembre de 1991 distribuye por email a un grupo de usuarios de Minix los fuentes de Linux.

Una de las grandezas de Linux consiste en que se trata de un operativo gratis, sus fuentes están distribuidos bajo la licencia de GNU, *GNU Public License* [2], la que permite que cualquier persona pueda modificar y/o usar los fuentes de Linux sin ningún cargo adicional.

La primera versión de Linux apareció en Internet el noviembre de 1991. Desde el primer momento se formó un grupo de programadores distribuido por todo el mundo, usando como único medio de intercomunicación la red Internet.

Las primeras versiones de Linux eran dificultosas de trabajar, poseían una documentación casi nula y no estaba recomendado para usuarios no avanzados. Actualmente existen varias distribuciones que automáticamente detectan el hardware y la instalación de paquetes se realiza con herramientas visuales.

Cualquier persona que considere poder realizar mejoras en el operativo puede incorporarse en el desarrollo de Linux. El diseño también es distribuido y se discute en varias *mailing lists* especializadas en ciertas partes de Linux.

3.1.1. Organización del capítulo

El capítulo de Linux no pretende explicar detalladamente el funcionamiento de todo el Linux, solo intenta presentar las características principales que posee. Se han incorporado descripciones detalladas del diseño interno de ciertas partes que se han necesitado en el desarrollo del proyecto.

[1] Esta guia no pretende explicar completamente el Kernel de Linux, solamente explicar la información básica necesaria para desarrollar un proyecto

[2] Ver anexo 11 para una descripción completa de la licencia de GNU

El capítulo se organiza en las siguientes áreas:

- La *introducción al Linux* 3.1, describe las principales características del Linux, así como un breve resumen histórico.
- *Introducción al Kernel* de Linux, 3.2, describe las principales estructuras del Kernel, así como las normas que se han de cumplir al diseñar nuevos módulos de Linux.
- Descripción del sistema de *Modules* de Linux en la sección 3.3, se explica como crear un module de Linux.
- Linux */proc* filesystem de la sección 3.4 muestra como crear entradas en el proc filesystem.
- Una descripción del *subsistema de red*, así como información necesaria para diseñar un device de red se encuentran en la sección 3.5.
- Explicación del funcionamiento de las *subsistema SCSI* de Linux, así como una descripción de los devices de SCSI.

3.1.2. Principales características

Linux soporta casi todas las características que se le puede pedir a un sistema operativo moderno, entre las más importantes hay que destacar:

Multi tarea , Linux es un sistema operativo multitarea real, existen módulos que permiten modificar la personalidad del scheduler y convertirlo en un pseudo Real-Time.

Multithreading , permite que varios flujos de ejecución compartan un mismo espacio de memoria.

Multiusuario , permite que varios usuarios utilicen simultáneamente el sistema.

Memoria Virtual pagina bloques de 4Kbytes a disco cada vez que necesita más memoria.

Cache dinámica ajusta el tamaño de los buffers a la memoria libre disponible en cada momento.

POSIX 1003.1 compatible.

Varios formatos de ejecutables soporta ejecutables en formato ELF y A.OUT.

Modo protegido opera en modo protegido del Intel, aunque existe en desarrollo una versión para 8088.

Varios sistemas de ficheros soporta Minix, ext2, msdos, umsdos (una versión extendida de msdos), hpfs (OS/2), sysv, SMB (sistema de ficheros de red en entornos Microsoft), FAT, FAT32, NCPFS (Novell), ufs, affs (Amiga FFS), romfs (sistema de ficheros para operar sobre EEPROMs) y el NFS de SUN.

Múltiples arquitecturas actualmente soporta alpha (Linux/Alpha), Intel386 (Linux/x86), PowerPC (Linux/PPC), MIPS (Linux/m4k), Motorola 68K (Linux/68k), Sparc (Linux/sparc), UltraSparc y Fujitsu AP1000 (con 64 procesadores).

Multiprocesador para las arquitecturas Intel, MIPS, Sparc y UltraSparc.

Múltiples protocolos de red , FrameRelay, X25, IPv4, IPv6, IPX, decnet, rose, lapb y appletalk. Actualmente se está trabajando intensamente en esta área.

Múltiples devices , casi todos los dispositivos hardware existentes en el mercado funcionan bajo Linux. Actualmente las propias casas diseñadoras de hardware trabajan en portar nuevas drivers a Linux a medida que salen nuevos equipos.

3.2. El Kernel

El Kernel de Linux posee una estructura bien definida, una vez se conoce un mínimo su estructura se puede encontrar cualquier parte con facilidad. En esta sección se trata de explicar la estructura del Kernel de Linux, así como las estructuras más importantes que obligatoriamente se han de conocer. No se comenta detalladamente como funciona pues evoluciona tan rápidamente que lo descrito un día no sirve en un par de meses. Sin embargo los cambios estructurales suelen distanciarse más en el tiempo, aunque también se producen con cierta periodicidad.

Antes de empezar con una descripción del Kernel se explica las normas y consejos existentes a la hora de retocar el Kernel de Linux.

3.2.1. Normas de codificación

Todo el Kernel de Linux está desarrollado en C, aunque existen numerosas líneas de código en ensamblador. *Siempre que sea posible se codifica en C, aunque alguna codificación en ensamblador fuera más óptima*. Solo se codifica en assembler lo imprescindible o aquellas funciones *bottlenecks* [3] que se ejecutan con asiduidad de forma que representan una perdida "considerable" en las prestaciones globales del kernel.

Una de las peores cosas que puede hacerse es estudiar y modificar el kernel en solitario, es decir sin contacto externo. Ha ocurrido muchas veces que varios grupos distribuidos por el mundo trabajan en proyectos similares pero su trabajo no se encuentra todavía en los kernels. En el mejor de los casos el problema suele "tratar" de solucionarse fusionando las dos partes, pero si alguno de ellos se fusiona un par de semanas antes que el otro, el nuevo proyecto ha de aportar mejoras "substanciales" para ser adaptado.

Antes de empezar a modificar cualquier parte del Kernel es aconsejable apuntarse a las mailing lists asociadas al código en el que se esta interesado. Las mailing lists discuten nuevos interfaces y se distribuyen versiones betas antes de fusionarse con la rama principal de Linux gestionada por Linus Torvalds.

Aunque el estilo de la codificación es muy personal, y no es obligatorio codificar de una forma en concreto existen unas normas en implícitas de estilo que a continuación se comentan.

- Siempre se ha de *inicializar las variables estáticas*, esta simple condición elimina muchos errores que se producen aleatoriamente.
- Si existe una publicación que explica el funcionamiento de la modificación a realizar, es aconsejable *indicar las referencias de donde se han obtenido nuevos algoritmos*, la publicación de donde procede.
- Indentar con *tabs de 8 espacios*, aunque mucha gente prefiere tabs de 4 espacios.
- Usar *un máximo de tres niveles de indentación*, si constantemente el código está indentado es mejor que se cambie la estructura de las funciones.

[3] Los cuellos de botella donde se pierde gran parte del tiempo de ejecución

- Utiliza *el estilo definido por Kernighan y Ritchie* para las llaves [4] con:

```
int function(int x)
{
        if (x is true) {
                we do y
        }
}
```

- La nomenclatura tendría que cumplir las siguientes características:
  - Las *variables temporales han de ser muy cortas* (tmp, val...).
  - En los contadores sin especial significación se acostumbra a usar i, j, k...
  - Solo se usan *variables globales si es estrictamente necesario*, en este caso el nombre de la variable ha de mostrar que información posee, por ejemplo *count_active_users*.
  - Codificar todas *las variables y funciones en Ingles*.
- Los comentarios son buenos, pero un **nunca** se ha de explicar **como** funciona el código, para eso ya existe el fuente. Se ha de explicar **qué** hace el código, pero sin excederse. En Linux está muy extendida la idea que no existe nada peor que un comentario falso, es decir cuando se realiza alguna modificación se ha de verificar que los comentarios existentes continuan siendo válidos.

3.2.2. Versiones de Kernel

Existen múltiples áreas de trabajo dentro de los fuentes de Linux, por un lado los grupos específicos de Linux evolucionan nuevas funcionalidades, y cuando consideran que ya son estables las envían a Linus Torvalds para que las fusione con la rama principal de Linux. Semanalmente suelen aparecer una media de dos versiones en la rama central de Linux, esto implica más de 100 versiones al año.

Actualmente, mayo de 1997, se está desarrollando la versión 2.1.XX, esta nueva versión incorpora notables mejoras en el entorno de red y reduce al mínimo la necesidad de botar la máquina para cambiar de configuración de kernel. Han existido grandes saltos en la evolución del kernel de Linux, normalmente se desarrolla en versiones impares y se estabiliza en las versiones pares, por ejemplo en la versión 1.3.X se incorporaron nuevas arquitecturas, pero hasta la versión 2.0.X no se dio por estable este diseño. Los grandes saltos en la evolución del Kernel han sido:

Linux 1.1.X Mayo 1994 - Marzo 1995

- Carga y descarga automática de módulos de kernel.

[4] Existe un programa, *indent -kr* , que retabula el código hasta dejarlo con el estilo de Kernighan y Ritchie

- Incorporación del NET3. Nuevo interface para el entorno de red.

Linux 1.3.X Marzo 1995 - Junio 1996

- Incorporación de nuevas arquitecturas.
- Soporte para arquitecturas multiprocesador.

Linux 2.1.X Junio 1996 - continua... (Mayo - 1997)

- Incorporación de la IPv6.
- Aumento de la granularidad del Kernel, "In-Kernel multithreading suppport".

Se puede observar que solo aparecen versiones impares, este es debido a que las versiones impares son las de desarrollo, mientras que las versiones pares corresponden a los fuentes estables (1.0.X,1.2.X,2.0.X ...).

3.2.3. Organización del Kernel

Antes de describir las estructuras más importantes del kernel de Linux, es necesario conocer como se estructuran los fuentes de Linux, así de donde se encuentran.

La localización habitual de los fuentes de Linux es el directorio */usr/src/linux*, aunque cualquier directorio es válido y solo suele trabajar se en este directorio por costumbre. A partir de ahora siempre que se nombre un fichero se referirá a partir del directorio base donde se encuentra la distribución de Linux.

Existe un gran nivel de estructuración, por ejemplo, la versión 2.1.33 posee 150 subdirectorios donde cada uno posee varios ficheros en C y/o assembler.

A un primer nivel de profundidad se encuentran los siguientes directorios,

Directorio	**Descripción**
/fs	Sistema de ficheros
/init	Inicialización del Kernel
/kernel	Gestión procesos
/lib	libc para el kernel
/mm	Gestión de la memoria
/include	todos los include
/net	protocolos de red
/ipc	IPC y *shared memory*
/drivers	drivers
/arch	Código dependiente de la arquitectura
/scripts	utilidades de compilación
/Documentation	Documentación

Cuadro 3.1.: Directorios a un nivel de profundidad

Cada uno de los directorios de la tabla 3.1 posee múltiples subdirectorios, a continuación se la utilidad de los directorios más importantes.

3.2.3.1. FileSystem /fs

Toda la gestión de ficheros se encuentra en este directorio, desde la carga de binarios, hasta los sistemas de ficheros soportados por Linux.

Posee un subdirectorio para cada sistema de ficheros que soporta, constantemente se aumentan los sistemas de ficheros soportados. Las últimas versiones incluso incorporan un "servidor" de ficheros NFS dentro de kernel. Los sistemas de ficheros soportados actualmente son los que aparecen en la siguiente lista:

- **/fs/msdos** Gestión de las particiones FAT
- **/fs/proc** proc filesystem 3.4
- **/fs/minix** Sistema de ficheros Minix
- **/fs/isofs** CDROM con formato ISO 9660
- **/fs/nfs** Network File System client
- **/fs/ext2** FileSystem de facto estándar en Linux
- **/fs/umsdos** Extensión de la FAT con nombres largos
- **/fs/hpfs** FileSystem del OS2
- **/fs/sysv** FileSystem del Coherent
- **/fs/smbfs** Cliente para redes SAMBA de Microsoft
- **/fs/fat** FileSystem MSDOS
- **/fs/vfat** FAT32 de windows95
- **/fs/ncpfs** Cliente NCP de Novell
- **/fs/ufs** FileSystem de SunOS4.x y algunos BSD
- **/fs/affs** FileSysmte del Amiga
- **/fs/romfs** Optimizado para FileSystems en memoria
- **/fs/autofs** Monta filesystems bajo demanda
- **/fs/nfsd** Server de NFS en kernel
- **/fs/lockd** Server para bloqueos de NFS

En este mismo directorio se encuentra toda la gestión de buffers de disco, quotas y el *Virtual File System* interface. El **VFS** se encarga de proporcionar un interfaz único a todos los sistemas de ficheros que existen en Linux, de esta forma es "fácil" crear un nuevo sistema de ficheros.

3.2.3.2. Inicialización /init

Cada vez que bota el kernel de Linux se bota el kernel de Linux se ejecuta la función *start_kernel* que se encuentra en *init/main.c*. Primero inicializa todas las CPUs que encuentra en el sistema, después inicializa la gestión de interrupciones, el scheduler y la *idle_task*. Una vez todo está operativo lee los parámetros que se le han pasado al kernel cuando ha botado [5].

3.2.3.3. Kernel /kernel

En el directorio */kernel* se encuentran todas las funciones independientes de la arquitectura que gestionan los procesos de Linux, desde el scheduler hasta la creación de nuevos procesos. Las funciones dependientes de la arquitectura se encuentran en */arch/XXX/kernel*, donde XXX corresponde a la arquitectura seleccionada.

3.2.3.4. Device Drivers /drivers

En */drivers* se encuentran todos los drivers de Linux, existe un directorio por cada tipo de driver, entre *./drivers/net* y *./drivers/isdn* se todos los drivers de gestión de red, *./drivers/block* posee todos los dispositivos de bloque (discos duros IDE principalmente), mientras que *./drivers/char* posee todos los dispositivos de caracteres (series, paralelos...), *./drivers/scsi* se encuentra toda la gestión de los dispositivos SCSI, las placas de sonido poseen los drivers de gestión en el directorio*./drivers/sound*, *./drivers/pci* la detección del bus PCI, *./drivers/cdrom* los CDROM que no son ni SCSI ni IDE, la detección del bus SBUS característico de máquinas SUN se encuentra en*./drivers/sbus*, en *./drivers/ap1000* se encuentra el hardware específico que posee el supercomputador Fujitsu, *./drivers/pnp* se encuentran los dispositivos Plug And Play, actualmente se está portando el puerto paralelo, y posiblemente en un futuro próximo se incluya el USB.

3.2.4. Estructuras más comunes

A continuación se explican las estructuras que se usan con más frecuencia a la hora de desarrollar en Linux. No pretende ser una descripción exhaustiva, solo una introducción a un mínimo a tener en cuenta. Hay que notar que en una versión normal de Linux existen más de 500 funciones exportadas [6] que pueden ser llamadas desde dentro del

[5] Mediante el lilo se pueden pasar parámetros al kernel en tiempo de boot
[6] La lista de funciones exportadas en Linux se encuentran el /proc/ksyms

Kernel. Estas funciones suelen estar agrupadas por áreas, por ejemplo el subsistema de SCSI posee 23 funciones exportadas.

A continuación se describen las funciones y estructuras más usadas por casi todos los subsistemas.

- Funciones y variables genéricas del Kernel.

  printk *int printk(const char *fmt, ...)*
  Hay que considerar que no es posible linkar los programas compilados para el kernel con la libc. Por este motivo existe la función *printk*, es el equivalente funcional al *printf* de la libc.

  kernel_thread *pid_t kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)*
  Crea un thread de kernel, el thread empieza a ejecutarse en la función *fn*, pasandole como parámetro el valor de *arg*, para un thread de kernel los flags han de ser cero. Para acabar con el thread simplemente ha de acabar la función que se ha llamado, *fn*.

  current *struct task_struct *current*
  current es un puntero a la tarea actual que se está ejecutando dentro de kernel. Suele utilizarse para programar dormirse sin consumir CPU durante un periodo de tiempo.

  ```
  current->state = TASK\_INTERRUPTIBLE;
  current->timeout = jiffies + 2; /* wait 20ms */
  schedule();
  ```

- Gestión de timeouts y clock del Kernel.

  jiffies *unsigned long jiffier*
  Contador del kernel que se incrementa cada vez que se produce una interrupción de reloj. Las incrementa cada 10ms aproximadamente. Hay mucha ocasiones que se utiliza esta variable para leer el "tiempo" en el que se está, también se suele usar la constante *HZ* que indica en cada arquitectura cuantas veces se incrementa *jiffies* cada segundo, en Linux/x86 100 veces == 10ms, mientras que en Linux/Alpha se incrementa 1024 veces por segundo.

  init_timer *void init_timer(struct timer_list *tlist)*
  Inicializa la estructura *tlist* para que pueda ser utilizada por las funciones *add_timer* y *del_timer*.

  add_timer *void add_timer(struct timer_list *tlist)*
  Ejecuta la función definida en *tlist→function* una vez ha pasado el tiempo especificado en *tlist→expires*. El valor de *expires* se calcula añadiendo a la variable *jiffies* el tiempo de timeout en 10ms. Por ejemplo *timeout = jiffies + 1;* esperaria 10ms antes de llamar a la *function*.

del_timer *int del_timer(struct timer_list *tlist)*
Des-programa el timer *tlist*.

- Principales funciones para gestionar las interrupciones.

cli *void cli()*
Inhibe las interrupciones.

sti *void sti()*
Permite que se produzcan interrupciones.

request_irq *int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long flags, const char *device, void *dev_id);*
Programa una interrupción, los parámetros de corresponden a:

- *irq* corresponde a la interrupción que intenta programar.
- Intenta programar la función *handler*. Cada vez que se llama la función *handler* pasa los siguientes parámetros:
  - Primer parámetro *int irq*, número de la interrupción que se ha producido.
  - Segundo parámetro el valor asignado en *dev_id* cuando se reserva la interrupción.
  - Como tercer parámetro un puntero al valor de los registros antes de que se produzca la interrupción.
- *flags*, las interrupciones rápidas se utiliza *SA_INTERRUPT*, las interrupciones largas usan 0 en el campo de flags.
- *device* en nombre del device que registra la interrupción.
- *dev_id* parámetro que se pasa a *handler* cuando se produce una interrupción.

free_irq *void free_irq(int irq, void *dev_id)*
Libera la interrupción *irq* siempre y cuando coincida el valor *dev_id* con el que se uso al registrar la interrupción.

- Funciones de entrada y salida:

inb *usigned int inb(unsigned short base)*
Lee un Byte del puerto localizado en *base*, los tres bytes superiores no son inicializados. También existe *inw* y *inl* que operan con 2 y 4 bytes respectivamente.

outb *usigned int outb(unsigned char val, unsigned short base)*
Escribe el Byte *val* al puerto localizado en *base*. También existe *outw* y *outl* que operan con 2 y 4 Bytes respectivamente.

- Gestión de la memoria en Linux

kmalloc *void *kmalloc(unsigned int size,int priority)*

Reserva un espacio de memoria de Kernel con una tamaño *size*. El campo *priority* puede poseer los siguientes valores:

GFP_BUFFER Busca un bloque de memoria de la lista de memoria libre, si no queda suficiente memoria no intenta paginar y retorna error.

GFP_ATOMIC Siempre que se reserva memoria desde una interrupción se ha de usar este flag. Fuerza a que no se produzca paginación.

GFP_KERNEL Busca un bloque de memoria, si no hay suficiente pagina y retorna un puntero al nuevo espacio de memoria.

GFP_DMA Todos lo bloques de memoria que necesiten operar con DMA han de reservarse con este flag. En Linux/x86 fuerza a que la memoria se encuentre entre los primeros 16Mbytes.

kfree *void kfree(void *block)*

Libera la memoria que previamente se había reservado con kmalloc.

get_free_page *unsigned long get_free_page(int priority)*

Reserva una página de memoria, el parámetro *priority* es igual al usado en *kmalloc*.

virt_to_phys *unsigned long virt_to_phys(void *addr)*

Convierte una dirección de virtual a física, es útil al programar hardware externo a la CPU para que opere con la memoria. Por ejemplo al programar una canal de DMA se le ha de pasar la memoria física.

phys_to_virt *void * phys_to_virt(unsigned long address)*

Función inversa a la anterior, a partir de la memoria física retorna un puntero a la memoria virtual a la que corresponde.

- Interacción entre espacio de Kernel y espacio de usuario.

verify_area *verify_area(int type, const void * addr, unsigned long size)*

Verifica que el espacio de memoria comprendido entre *addr* y *addr+size* es del tipo *type*. *type* puede tener dos valores, *VERIFY_READ* y *VERIFY_WRITE*. Ha de usarse en devices cuando se han de pasar datos al espacio de usuario.

copy_to_user *int copy_to_user(void *to, const void *from, unsigned long n)*

Copia el bloque residente en el espacio de kernel al espacio espacio de direcciones de usuario. Verifica que se pueda escribir en el espacio de usuario.

copy_from_user *int copy_from_user(void *to, const void *from, unsigned long n)*

Copia el bloque residente en el espacio de usuario al espacio espacio de direcciones de kernel. Verifica que se pueda leer del espacio de usuario.

- Gestión de semáforos.

down *void down(struct semaphore *sem)*

Decrementa el valor de *sem→count*, si posee una valor negativo bloquea al proceso. Mientras el proceso este dormido no puede ser matado.

down_interruptible *int down_interruptible(struct semaphore *sem)*
Al igual que *down* decrementa el valor de *sem→count*, si el valor resultante es negativo duerme el proceso, pero a diferencia de *down* el proceso puede recibir signals, es decir puede matarse. Si ha recibido un signal retorna menor que cero, si se ha despertado mediante un *up* retorna cero.

up *void up(struct semaphore *sem)*
Incrementa el valor de *sem→count*, si es positivo y hay un proceso dormido asociado, despierta un proceso que previamente se había dormido con *down*. Es posible realizar varios *up* seguidos de forma que incrementa el valor de *count*, para dormir el proceso será necesario realizar varios *down*.

- Operaciones atómicas.

set_bit *int set_bit(int nr,void *addr)*
Esta instrucción atómicamente cambia a cierto el bit *nr* del contenido de la dirección *addr*, retorna el valor que poseía antes de realizar el cambio.

clear_bit *int clear_bit(int nr,void *addr)*
Esta instrucción atómicamente cambia a falso el bit *nr* del contenido de la dirección *addr*, retorna el valor que poseía antes de realizar el cambio.

atomic_dec_and_test *int atomic_dec_and_test(atomic_t *v)*
Atómicamente decrementa el contenido de la dirección de *v*, hay que destacar que ha de ser de tipo *atomic_t*. Compara el resultado de decrementar con cero, de forma que retorna cierto si es distinto de cero después de decrementar.

3.2.5. Compilación

Para compilar los fuentes del kernel se utiliza el Makefile de GNU y el compilador GCC, teóricamente cualquier compilador ANSI-C tendría que funcionar, en realidad algunas personas han utilizado el compilador lcc aunque fue hace varios meses.

La compilación del Kernel se realiza por fases, configuración, compilación del kernel, compilación de los modules.

3.2.5.1. Configuración del Kernel

Con la intención de listar todas las posibles opciones dependiendo de la máquina de la que se dispone se utiliza la utilidad *configure*, aunque para compilar el kernel no es necesario conocer su funcionamiento.

En un principio para configurar el kernel solo existía la opción:

```
#make config
```

Actualmente existe un entorno gráfico de X-Windows, construido en TCL-TK que permite configurar el kernel como muestra la figura 3.1, para invocarlo se ha de ejecutar:

```
#make xconfig
```

También existe un entorno de menús usando los ncurses, para invocarlo se usa:

```
#make menuconfig
```

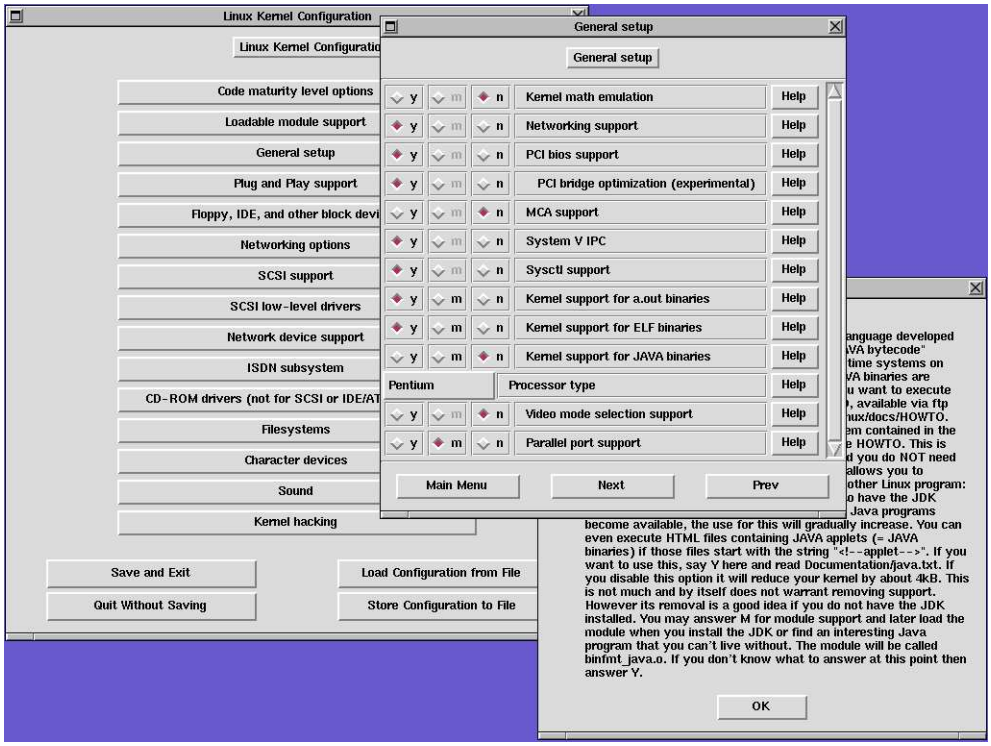

Figura 3.1.: Configuración mediante xconfig

En los tres programas de configuración se elije cada parte del kernel como se desea compilar, *Module*, *In-Kernel* o *None* cuando no se desea compilar. No todas las partes del kernel se pueden compilar como modules y de hecho las partes del kernel que son necesarias para botar la máquina no han de compilarse como modules. Por ejemplo si tenemos un disco SCSI y una placa de red, no se puede compilar como module el subsistema SCSI pues seria imposible botar, no obstante si se puede compilar como module la placa de red pues no es necesaria para botar la máquina.

En cualquier momento durante la configuración se puede pedir más información sobre cualquier opción mediante la tecla h (Help). Toda esta información esta en /Documentation/Configuration.help. Se ha de recordar que al crear un nuevo module de Linux se ha de crear una entrada en este fichero.

3.2.5.2. Compilación kernel y modules

Cada vez que se modifican la configuración del kernel es necesario regenerar las dependencias. Esto se realiza mediante el comando:

```
#make depend
```

Una vez se han generado los dependencias ya es posible compilar el kernel mediante:

```
#make zImage
#make modules
```

La primera linea generará una versión comprimida del kernel que se puede instalar mediante el lilo. La segunda linea compilará todos los modules y creará links en el directorio */modules*.

Mientras se desarrolla un nuevo module no es necesario recompilar todos los modules cada vez que se realiza una modificación. Si se esta crearon un device de red */drivers/net* es posible usar la siguiente linea para recompilar solo los devices de red.

```
#make SUBDIRS=drivers/net modules
```

En realidad hay mucha gente que trabaja desde el emacs o xemacs para codifica nuevos devices . Desde el mismo emacs se genera solo la linea de gcc necesaria. No obstante se ha de tener en cuenta que las opciones de compilación suelen variar con el tiempo. Por este motivo lo mejor es compilar con el sistema tradicional y después realizar un "cut-and-paste" de la linea de gcc para que el emacs solo ejecute esta linea. De esta forma con un simple C-cC-c se recompila el modulo sobre el que se trabaja.

3.2.5.3. Añadir nuevos modules

Si se desea que al añadir un nuevo modulo al kernel salga en los programas de configuración, tenemos que añadir algunas entradas en los ficheros *Config.in* y *Makefile* que aparece en el directorio donde reside el nuevo module.

Por ejemplo supongamos que se crea un nuevo device de red llamado *thebest*, en este caso el código se encuentra en /drivers/net. En el fichero */drivers/net/Config.in*, se ha de añadir la entrada del nuevo device:

```
tristate 'The Best Network Device' CONFIG\_THEBEST
```

Ahora cuando se configure los devices de red, preguntara si deseamos incluir *The Best Network Device*, no obstante para que realmente se compile se ha de añadir algunas lineas a fichero */drivers/net/Makefile*.

```
ifeq ($(CONFIG\_THEBEST),y)
L\_OBJS += thebest.o
else
  ifeq ($(CONFIG\_THEBEST),m)
  M\_OBJS += thebest.o
  endif
endif
```

Las lineas anteriores verifican el valor de la variable CONFIG_THEBEST, compilándolo como module M_OBJS o como librería L_OBJS, dependiendo de la opción elegida al configurar el kernel.

Si el proyecto fuera de mayor embergadura y estuviera dividido en varios ficheros C que luego se han de linkar las cosas se complican un poco más. Un ejemplo donde puede observarse como hacerlo se encuentra en */drivers/pnp*.

3.3. Los Modules

El kernel de Linux ha crecido constantemente desde sus inicios. Las primeras versiones incorporaban los dispositivos mínimos, y pocas funcionalidades extras. Actualmente incorporan gran cantidad de dispositivos, varios protocolos de red, múltiples sistemas de ficheros ... Solo hay que considerar que los fuentes de la versión 1.0 de Linux ocupaba 5Mbytes mientras que las últimas versiones 2.1.33 ocupan más de 30Mbytes, en esta versión existen más de 900.000 lineas de código, y al ritmo de crecimiento actual a principios de 1998 se pasará del millón de lineas de código.

Todos los operativos monolíticos (la versión 1.0 de Linux era monolítica) poseen varios problemas de diseño. Una característica remarcable consisten en la necesidad de recompilar *todo el kernel* cada vez que se modifica algún dispositivo, también es necesario rebotar la máquina con el nuevo Kernel. Por otro lado el kernel incorpora todos los dispositivos, aunque no se utilicen, ocupando la memoria de la máquina.

Las primeras versiones de Linux eran monolíticas tradicionales, a los problemas ya comentados, se ha de añadir a la dificultad de desarrollar nuevos drivers, cada vez que se realiza una modificación es necesario rebotar toda la máquina. Con la intención de solucionar todos estos problemas se crearon los **modules** de Linux.

3.3.1. Que son los modules

Desde el punto de vista del Kernel se podría definir module como, *un objeto objeto que pueden cargarse y descargarse del kernel en tiempo de ejecución.*

Los modules se cargan con los mismos derechos que posee el kernel, es decir comparten el espacio de direcciones del Kernel, y pueden ejecutar instrucciones privilegiadas. A diferencia de varios microkernels, los modules pueden interferir al resto del sistema.

Otra ventaja de desarrollar los dispositivos consiste en la "limpieza" del interfaz, cada module se asemeja a un objeto con funciones privadas y funciones públicas, de forma que solo las funciones públicas pueden ser accedidas por el resto del Kernel.

El resumen de las ventajas que aportan los modules de Linux son:

- *Ocupación de memoria:* Reducen el tamaño ocupado por el Kernel de Linux en memoria, solo se cargan los subsistemas necesarios
- *Facilidad de desarrollo:* No es necesario recompilar todo el Kernel cada vez que se modifica una parte.
- *Claridad de interfaces:* Se definen los puntos de entrada posibles en cada module, de esta forma se consigue un diseño basado en objetos, con funciones públicas y funciones privadas.

3.3.2. Otras aproximaciones

Existe una tendencia a volver a incorporar ciertos subsistemas dentro de kernel. En un principio existían los Kernels monolíticos con todo dentro del Kernel, después

con la aparición de los microkernels se crearon servidores para cada dispositivo, pero actualmente tratando de mantener cierto nivel de garantías se tiende a incorporar los dispositivos a Kernel con la intención de aumentar prestaciones.

En los últimos años se observan soluciones al problema de los servidores en espacio de direcciones de los operativos con microkernel que se aproximan a los diseños monolíticos. Últimamente también ha aparecido los Exo-Kernel que proponen una cambio radical en el diseño con la intención de conseguir máximas prestaciones y mantener un diseño con protección entre objetos.

Existen varias alternativas en la incorporación de funcionalizadades a nivel de Kernel. A continuación se comentan las más destacables.

3.3.2.1. Windows NT™

Windows NT™ [Bak96] posee una estructura de carga y descarga de objetos similar a la utilizada en Linux. Los objetos de kernel pueden cargarse al botar, cuando se inicializa el kernel o simplemente cuando son necesarios. Al igual que Linux los dispositivos pueden descargarse en tiempo de ejecución y ser reemplazados por nuevas versiones.

La versión 3.51 posee muchos servicios que funcionan en el espacio de direcciones de usuario, de forma que un error no pueden afectar al resto del operativo. No obstante esta versión aunque es muy apreciada académicamente posee problemas de prestaciones. En la versión 4.0 muchos de estos servicios se han incorporado en el espacio de Kernel, de forma que pueden acceder a las instrucciones privilegiadas. Un ejemplo lo supone el CSRSS o Win32 Server que se encarga de la gestión del GUI. Con la incorporación del CSRSS a espacio de Kernel se ahorran ciclos necesarios para pasar mensajes, pero aumenta el riesgo a que un error pueda saturar todo el sistema.

3.3.2.2. SPIN

El sistema SPIN [SFPB95] también permite la carga y descarga de modules del espacio de kernel, pero a diferencia que el NT y el Linux, lo hace de forma segura. Para conseguir este objetivo fuerza a que el operativo se desarrolle en Modula-3, y cada vez que carga un module chequea que no accede fuera de límites.

Esta solución garantiza que los modules no afectan al resto de sistema y mantiene la prestaciones de un operativo monolítico tradicional, pero fuerza a que los diseños se realicen en Modula-3.

3.3.2.3. INKS - IN Kernel Server

Actualmente el microkernel Mach, que siempre había estado centrado en servidores en espacios de direcciones distintos, también esta tendiendo en esta dirección. El proyecto INKS [LHFL95] de la Universidad de Utah permite que modules que han sido testeados en espacio de usuario puedan incorporarse a espacio de kernel con la intención de aumentar las prestaciones.

3.3.3. Como se gestionan

Existe un conjunto de herramientas que permiten operar con los modules de Linux. Estas herramientas se encuentran en todas las distribuciones actuales de Linux, pero a causa de la continua evolución es conveniente asegurarse que se dispone de una versión apropiada [7].

Los modules se almacenan en el directorio */lib/modules*, creando un subdirectorio por cada versión de kernel existente. Dentro del subdirectorio de versión existe un fichero *modules.dep* que posee un listado de las dependencias entre modules. Este fichero se genera automáticamente mediante la utilidad depmod.

Al cargar un module añade las funciones o variables por las que puede ser accedido al fichero */proc/ksyms*. En este fichero están todas las funciones con las que se pueden linkar los modules cuando se cargan a espacio de kernel.

Las principales utilidades son:

- **insmod**, carga los modules al espacio de direcciones de kernel, permite que se le pasen parámetros de carga. Por ejemplo *insmod ne base=0x300*, fuerza a que el module de red para placas NE2000 opere en la dirección base 0x300.

- **rmmod**, descarga los modules de Kernel. Por ejemplo *rmmod ne* descarga el gestor de red. Solo permite descargar los modules que no están siendo utilizados por otros modules.

- **lsmod**, muestra los modules cargados en el sistema, por cada module cargado informa del tamaño en Bytes que ocupa y por que otros modules esta siendo referencia-do.

- **kerneld**, ejecutado como un daemon carga automáticamente los modules que son necesarios, por ejemplo si los dispositivos de ne2000 se compilan como module, cuando se realice un ifconfig de la placa de red, automáticamente cargara el module *ne*. Este daemon facilita la gestión de los modules, después de configurar el fichero /etc/conf.modules con las dependencias de modules carga todos los modules necesarios automáticamente.

- **depmod** genera las dependencias entre los modules, normalmente se ejecuta cada vez que bota el kernel.

3.3.4. Filosofia de trabajo

El diseño aplicado a los módulos de un determinado nivel está pensado para ser lo mas sencillo, flexible y claro que se pueda. De esta manera, unas APIs bien claras entre los módulos inferior y superior permiten una mejor y más rápida creación o modificación, para nuevos tipos de APIs.

[7] El documento /Documentation/modules.txt especifica cual es la versión necesaria de modutils para kernel actual

Cada módulo define una serie de APIs de interacción con sus niveles colindantes, de manera precisa y a modo de pre/post condiciones. De esta manera se encapsula en funcionamiento interno, y se da una API común y homogénea para los módulos que interaccionan.

Existen tres tipos de funciones con las que se trabaja:

Funciones exportadas Son las funciones que implementa un módulo, pero que pueden ser llamadas desde otros. Son los puntos de entrada o sincronismo que utilizan los otros módulos para interactuar con el módulo que las exporta.

Funciones importadas Son las que utiliza un módulo para comunicarse con funciones de otro. Estas deben ser convenientemente exportadas por el otro módulo. Cabe decir que el concepto de "importar" no existe, sino que significaría sencillamente poder utilizar funciones exportadas por otros.

Funciones internas Son todas aquellas funciones internas del módulo que nadie puede llamar a excepción de el mismo.

Se debe tener en cuenta que las funciones exportadas no solo pueden utilizarse con el mecanismo de *modules* visto en la sección 3.3. Una operación que puede dar a un módulo las facilidades de exportar funciones puede ser pasar una función (puntero) como parámetro, a otro módulo, mediante una función exportada por el método de *modules* comentado. De esta manera le podemos hacer llegar a un module, una referencia a una función que podrá llamar a partir de este momento.

Este tipo de mecanismos, se usan por ejemplo para pasar una serie o estructura de funciones fija y definida para la interacción de los dos.

El sistema de registro entre módulos, es decir cuando se intercambian esta información para prepararse para su interacción, se hace siempre de manera que los módulos de un cierto nivel activan las funciones de registro de su nivel superior, que a la vez hará lo mismo con su nivel superior... Con este sistema los módulos que deben cargarse por orden y empezando por los niveles más altos.

3.3.5. Implementación

Por el momento se ha comentado como funcionan los modules y como se puede operar con ellos, en esta sección se explicará que consideraciones especiales se han de tener al diseñar modules en Linux.

3.3.5.1. Un ejemplo de module

La forma más sencilla de ver que es necesario para crear un module es crear un ejemplo lo más sencillo posible que muestre las posibilidades de los modules.

Como ejemplo crearemos un module que exporte una función, se linke con varias funciones de kernel y exporte una nueva función para que pueda ser utilizada por otros modules.

El makefile necesario para nuestro ejemplo es el siguiente:

```
CFLAGS=-D\_\_KERNEL\_\_ -Wall -O2 -DMODULE -DEXPORT\_SYMTAB
LDFLAGS= -O2

all:    test.o

clean:
        rm -f *.o
```

El fuente, test.c ha de poseer como mínimo los siguientes includes:

```
#include <linux/module.h>
#include <linux/delay.h>
#include <linux/kernel.h>
#include <linux/malloc.h>
#include <linux/utsname.h>
```

El include module posee todas las macros necesarias para operar con modules. En las últimas versiones de modules es posible incluir información acerca de los autores mediante la macro **MODULE_AUTHOR**, en nuestro ejemplo:

```
MODULE\_AUTHOR(``Jose and Jose'');
```

Con la intención de aceptar parámetros a la hora de cargar el module se ha de utilizar la macro **MODULE_PARM**. Esta macro permite indicar el tipo de parámetro. En nuestro ejemplo se permite que en tiempo de carga se modifique el valor de irq, si no se pasa ningún parámetro mantendrá el valor de 7, a continuación se ve la declaración realizada:

```
int irq=7;
MODULE\_PARM(irq);
```

Con la intención que otros modules puedan reverenciar funciones utilizadas por nuestro module existe la macro **EXPORT_SYMBOL**, a esta macro se le pueden pasar tanto punteros a funciones, identificadores de variables. En nuestro caso incorporará a la lista de funciones públicas la función test_function.

```
void test\_function( void )
{
  printk("test\_func kernel[%s]\n",system\_utsname.sysname);
}
EXPORT\_SYMBOL(test\_function);
```

Por norma todas las funciones que exportan los modules suelen empezar con el nombre del module y después un underline, en nuestro ejemplo como el module se llama test todas las funciones deberían empezar por test_.

Se han de declarar las funciones asociadas a la cargar y descargar del module, en el ejemplo quedaría:

```
int init\_module( void )
{
  printk("init\_module\n");

  printk("test\_func kernel[%s]\n",system\_utsname.sysname);
  printk("irq [%d]\n",irq);
  return 0;
}

void cleanup\_module( void )
{
  printk("cleanup\_module\n");
}
```

La función **init_module** se llama cuando se carga el module, dentro hacemos referencia a la función printk y a la variable system_utsname que son exportadas por otros modules.

Antes de descargar el module se llama a la función **cleanup_module**, esta función ha de liberar todos los recursos que ocupa el module.

Existen dos macros que no se han utilizado en el ejemplo anterior, pero que son muy útiles a la hora de trabajar con los modules. Cada module posee un contador que puede incrementarse con la macro **MOD_INC_USE_COUNT**, y puede decrementarse con **MOD_DEC_USE_COUNT**, mientras que el contador sea distinto a cero el module no podrá descargarse. Se acostumbra a utilizar un INC en las funciones de open y un DEC en los close, de esta forma cuando el contador está a zero se puede saber si el module no esta usado por nadie y puede descargarse temporalmente.

3.3.6. Carga y descarga por parte del kernel

El kernel proporciona tres llamadas a sistema que se encargan de cargar los modules:

- *create_module*, reserva un espacio de memoria en el área de kernel donde se cargará el module. Acepta como parámetros el nombre del module y el espacio de memoria necesario para el module.
- *init_module*, inicializa un module en el espacio de kernel, llamando a la función *init_module* que se ha de definir en el module de Linux. Una vez esta inicializado añade en la lista de símbolos, /proc/ksyms, las funciones que exporta el module cargado.
- *delete_module*, ejecuta la función *cleanup_module* definida en el module y después descarga el module de memoria.

Las tres funciones que proporciona el kernel para operar con los modules son utilizadas por las utilidades insmod y rmmod ver la sección 3.3.3 en la pagina 57 para ver como funcionan estos comandos.

3.3.7. Conclusiones

Los modules de Linux han permitido que el kernel pueda incorporar funcionalidades que solo se cargan cuando son necesarias, también han facilitado un diseño más modular. Muchos cambios que se han realizado el los subsistemas de Linux se han originado a la necesidad de definir interfaz claras entre diferentes modules.

Actualmente casi todos los subsistemas de kernel pueden cargarse y descargarse en tiempo de ejecución, de forma que se consigue un kernel altamente especializado a las necesidades de la máquina donde se ejecuta.

3.4. El sistema de ficheros /proc

Un sistema de ficheros especial consideración a la hora de diseñar módulos de kernel en Linux es el Proc filesystem. Al igual que todos los sistemas de ficheros se implementa sobre el **VFS** de Linux, aunque posee un uso muy especial.

Al contrario que el resto de sistemas de ficheros, el *Proc* no pretende almacenar información para después poder ser procesada, más bien permite una relación directa entre el sistema de ficheros y el kernel de Linux. En cualquier distribución de Linux se monta en /proc, y se puede acceder como si fuera otro sistema de ficheros, pero solo hacer un dir en */proc* podemos observar como la información varía de dir en dir. Esto es debido a que muestra la información dinámica del Kernel de Linux.

3.4.1. Principales entradas

A primera vista se observa que existe un subdirectorio por cada proceso existente, un conjunto de ficheros que informan del estado de la máquina y unos cuantos subdirectorios específicos.

Todas las personas que pretendan modificar partes del Kernel de Linux han de conocer como mínimo los siguientes ficheros:

/proc/ioports Contiene una lista de los espacios de memoria registrados en el kernel. Cada vez que se encuentra un nuevo dispositivo hardware se ha de reservar mediante la función *request_region* el mapeo de memoria que utiliza. Por ejemplo una placa de red podría estar mapeada entre los puertos 0x300 y 0x31f.

/proc/interrupts Indica cada una de las interrupciones del sistema por que dispositivo esta reservada.

/proc/ksyms Posee el punto de entrada de todas las funciones exportadas por el kernel. Cada vez que se carga un module que exporta funciones para que puedan ser usadas por otro module se muestran en esta lista.

/proc/sys Este directorio de reciente creación posee los parámetros configurables de kernel. Normalmente se pueden modificar solo como root, pero eso depende de los permisos de modificación del fichero. Por ejemplo se puede activar el routing del kernel realizando un:

```
echo 1 >/proc/sys/net/ipv4/ip\_forwarding
```

De igual forma para desactivar que el Linux actúe como router simplemente se ha de realizar:

```
echo 0 >/proc/sys/net/ipv4/ip\_forwarding
```

Existen muchas otras entradas que son muy útiles, aunque no se comenten en este documento.

3.4.2. Creación de una entrada

Normalmente los subsistemas que muestran información mediante el *Proc* crean una entrada durante la inicialización y eliminan la entrada al descargarse el módulo.

Últimamente se han producido algunos cambios en la generación de directorios y ficheros en el *Proc*. Actualmente para crear un entrada se ha de rellenar la estructura *struct proc_dir_entry*, los campos necesarios son:

data Puntero a la estructura que se pasará como parámetro cada vez que se llame a *read_proc* y a *write_proc*.

read_proc *int read_proc(char *page, char **start, off_t off, int count, int *eof, void *data)*
Cada vez que se lee una el contenido de la entrada a crear se llamará a la función indicada. Ha de retornar la cantidad de bytes leídos en cada lectura.

- **page* Se pasa a la función una página de tamaño PAGESIZE, donde se ha de copiar el valor a mostrar. Si no hay suficiente con una página el valor de **eof* ha de ser falso.
- ***start* cuando se usa menos de una página se ha de retornar **start = 0*.
- *off* informa de cuanto se ha leído. La primera vez vale 0, la segunda tamaño que se haya leído...
- *count* El espacio de página que se puede utilizar. Siempre es inferior a PAGESIZE.
- **eof* El valor de **eof* ha de ser cierto una vez se finalice el proceso de lectura.
- **data* Se pasa el valor que se ha almacenado en *proc_dir_entry→data* antes de registrar la entrada.

write_proc *int write_proc(struct file *file, const char *buffer, unsigned long count, void *data)*
Cuando se intenta modificar el fichero creado en *proc_dir_entry* se llama a la función apuntada por write_proc. Ha de retornar la cantidad de bytes que ha aceptado en la escritura. Cada unos de los parámetros corresponde:

- **file* Puntero a la estructura *file* a la que corresponde el fichero seleccionado.
- **buffer* apunta a los valores que se han pasado para modificar la entrada.
- *count* indica el tamaño que posee *buffer*.
- **data* Se pasa el valor que se ha almacenado en *proc_dir_entry→data* antes de registrar la entrada.

Una vez se ha rellenado la estructura correctamente se ha de llamar a la función *create_proc_entry*, esta función necesita como parámetros:

- *char *name*, un puntero al nombre que ha de poseer la entrada, por ejemplo "pnp" crearía una entrada a partir de donde se encuentre *parent*. Hay que tener en cuenta que no se reserva memoria para el nombre de forma que ha de ser estático el contenido de name.

- *mode_t mode*, se utiliza para especificar los permisos que ha de poseer el fichero. Los más frecuentes son:

  S_IFDIR Indica que se trata de un directorio.

  S_IFREG Cuando se crea un fichero.

  S_IRUGO Permisos de lectura para todos, propietario, grupo y others.

  S_IXUGO Permisos de ejecución o paso para todos, propietario, grupo y others.

  S_IWUSR Permiso de escritura para el propietario.

- *struct proc_dir_entry *parent*, cada vez que se crea una entrada en Proc se ha de indicar con respecto a quien se crea. Por ejemplo si queremos crear una entrada en */proc* se ha de pasar como parámetro *proc_root*.

Si se desea eliminar una entrada ya creada se ha de llamar a *proc_unregister* pasando como primer parámetro el valor usado como base en *create_proc_entry*, y como segundo parámetro el valor de *low_ino*de la estructura retornado en *create_proc_entry*. Un ejemplo seria:

```
proc\_unregister(&proc\_root,proc\_dir\_entry\_value->low\_ino);
```

3.5. La Red

Desde un comienzo el Linux disponía de entorno de red TCP/IP, las primeras versiones NET1, no llegaron a aparecer en versiones estables de Linux. En la versión 1.0 de Linux ya se encontraba el entorno NET2. En Linux 1.2, se integró la versión actual más conocida como NET3.

Al igual que gran parte del trabajo de las estructuras principales del kernel corresponden a Linus Torvalds, también es bien cierto que Gran parte del trabajo del entorno red ha sido desarrollado por Alan Cox.

Actualmente se los principales protocolos operativos son:

IPv4 El TCP/IP usado en Internet, corresponde a la versión 4. Se encuentra disponible desde la primera versión de Linux.

IPv6 Más conocido como IP6 o IPng, IP Next Generation, es una evolución sobre IPv4, soporta encriptación de paquetes, y opera con un espacio de direcciones de 16 Bytes.

Amateur Radio x25 Conexión punto a punto mediante dos equipos de radio.

IPX Soporta el protocolo de Novell desarrollado para sus redes locales.

appletalk Protocolo usado por ordenadores apple para encontrarse dentro de una misma red.

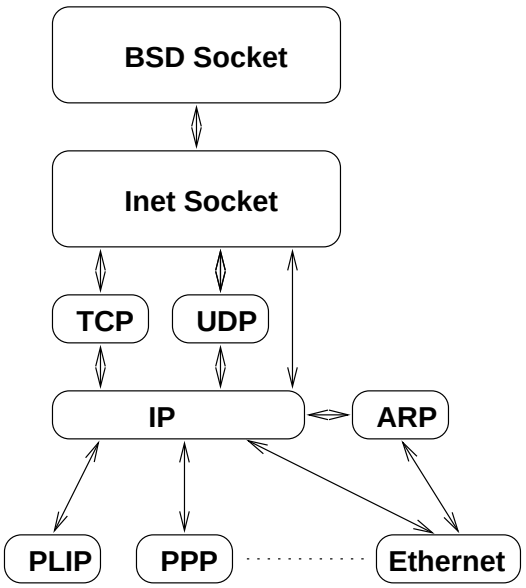

Figura 3.2.: Estructura parcial de módulos de red

Antes de empezar a explicar como funciona el entorno de red en Linux, se ha de conocer donde se encuentran el código que gestiona la red. En */net* se encuentra los protocolos de alto nivel, y los drivers de red están en */drivers/net*.

Con solamente cinco estructuras [8], se puede entender casi completamente el funcionamiento genérico del entorno de red, y ser capaces de desarrollar nuevos device drivers.

socket Implementación básica de los sockets de BSD

sock Conocida como el socket de INET o NET3.

proto Los protocolos TCP, UDP y otros se acceden mediante una estructura abstracta creada por cada protocolo y almacenada en la estructura *proto*.

sk_buff Gestión de los paquetes que se envían y reciben en cualquier protocolo de red.

device Creada por el device driver de red para registrarse en el kernel, es usada por este para operar con el device.

3.5.1. La estructura sk_buff

Los paquetes son gestionados mediante la estructura *sk_buff*. Esta estructura permite un paso de información entre las distintas capas del kernel sin necesidad de realizar una copia de la información. Existe un convenio de nomenclatura en el cual se acostumbra a llamar *skb* a cualquier puntero de *sk_buff*.

Existe un conjunto de funciones asociadas a los *skb*, las principales son:

dev_alloc_skb *struct sk_buff *dev_alloc_skb(unsigned int length)*
Crea un nuevo *skb*, son creados tanto por los protocolos de alto nivel, como por los devices cada vez que se recibe un nuevo mensaje.

dev_kfree_skb *void dev_kfree_skb(struct sk_buff *skb, int rw)*
Libera un *skb*, cada vez que se libera se ha de indicar si el skb era para enviar (creado por un protocolo de alto nivel) o para recepción (creado por el device de red), se usa FREE_WRITE y FREE_READ respectivamente.

skb_push *char *skb_push(struct sk_buff *skb, unsigned int len)*
Reserva espacio en la cabecera del paquete. Tiene que existir suficiente espacio desde que se creo el skb.

skb_put *char *skb_put(struct sk_buff *skb, unsigned int len)*
Reserva espacio en la cola del paquete. Tiene que existir suficiente espacio desde que se creo el skb.

[8] En la versión *Linux 2.1.40*pues podría producirse un cambio

Cuando se recibe un nuevo mensaje se ha de crear un *skb* mediante la función *dev_alloc_skb*. Una vez creado se reserva espacio al final del buffer con *skb_put* y se copia la información recibida, antes de pasar la información al nivel superior se ha de rellenar el protocolo al que corresponde, *protocol*, el device que esta procesando la recepción *dev* [9] e inicializar el puntero a la cabecera.

```
/* Interrupt con datos en *data y tamaño en count */
void incoming\_message(char *data, int count)
{
sk\_buff *skb = dev\_alloc\_skb (count);

memcpy(skb\_put(skb,count), data, count);

/* puntero a la estructura device con la que se creo el device */
skb->dev      = device;
skb->protocol = proto;
skb->mac.raw  = skb\_data(skb);

netif\_rx(skb);
}
```

Cuando se envía un mensaje el device a de enviar los datos que residen en *data* al dispositivo. Una vez han sido enviados correctamente se ha de llamar a *dev_kfree_skb* pasando como parámetros el puntero al skb y FREE_WRITE para indicar que se a enviado en skb.

```
/* Envía mensaje */
int hard\_start\_xmit(sk\_buff *skb, struct device *dev)
{

if( skb->protocol == ETH\_P\_xxx )
  Envia\_Fisicamente\_proto\_xxx(skb->data,skb->len);

dev\_kfree\_skb (skb, FREE\_WRITE);
}
```

Antes de enviar el mensaje es conveniente verificar que los punteros son distinto a null.

3.5.2. La estructura device

La *struct device* se encuentra en *include/linux/netdevice.h*, cada *Network Device* de red ha de rellenar esta estructura en tiempo de inicialización.

[9] ver la estructura device para más información

Actualmente esta estructura esta catalogada por el propio kernel como un gran error, posee desde información de protocolos de alto nivel hasta puertos de entrada y salida de los dispositivos hardware que gestiona.

La *struct device* esta compuesta por una serie de variables y punteros a funciones que ha de rellenar cada device que se crea.

Nombre del device *char *name*
Nombre del device de red, el device de red ha de encargarse de la gestión de la memoria necesaria para almacenar el nombre.

direcciones de entrada salida actualmente se puede almacenar dos bloques de memoria, junto con la dirección base para IO y la IRQ utilizada. Claramente esto ha de cambiar y no seria de extrañar que en próximas versiones se utilice la nueva estructura *ifmap*.

Flags del dispositivo Existen tres flags que indican el estado del dispositivo de bajo nivel.

start Se activa cuando existe una operación en curso, solo sirve de información para el *Network Device*, pues puede ignorar su valor.

interrupt Durante el periodo que se gestiona una interrupción este ha de estar activado. Al igual que *start* solo es utilizado por el device de red.

tbusy Cuando el device de red no puede procesar más peticiones, activa este valor de forma que los protocolos de alto nivel no llamaran de nuevo a la función.

next Todos los devices que han sido correctamente inicializados se encuentran en una lista lineal.

ifindex Identificador único de cada device de red, es usado por algunos protocolos como identificador en funciones de hash.

next_up Lista de todos los dispositivos de red que se encuentran UP.

flags Los flags del interface al estilo BSD. Todos los parámetros que suelen gestionarse desde el *ifconfig*.

IFF_UP El interface esta en marcha.

IFF_BROADCAST Permite direcciones de broadcast.

IFF_DEBUG Activado los mensajes de debug.

IFF_LOOPBACK Es el interface de loopback.

IFF_POINTOPOINT Es de punto a punto, con dirección origen y destino.

IFF_NOTRAILERS Evitar el uso de *trailers*.

IFF_RUNNING Posee recursos reservados.

IFF_NOARP No soporta ARP.

IFF_PROMISC Opera en modo promiscuo, es decir acepta todos los paquetes de la red.

IFF_ALLMULTI Recibe paquetes multicast.

IFF_MASTER El device actual es el master cuando existen varios devices de red para un mismo canal de comunicación.

IFF_SLAVE El device actual es el master cuando existen varios devices de red para un mismo canal de comunicación.

IFF_MULTICAST Soporta multicast.

family Tipo de direcciones que soporta, normalmente se usa AF_INET, para ver la lista completa de familias disponibles es necesario visitar el include *linux/socket.h*.

mtu Tamaño máximo de trama aceptada por el device de bajo nivel.

type Tipo de cabecera que se utiliza, por ejemplo ARPHRD_ETHER corresponde a las cabeceras Ethernet.

hard_header_len Tamaño de la cabecera que se utiliza.

priv Puntero a void donde cada device de red suele almacenar toda la información local que necesita para operar.

broadcast Dirección broadcast que usa el dispositivo, solo es necesario rellenarla cuando *flags* incluye *IFF_BROADCAST*.

dev_addr Dirección hardware del device de red.

mc_list Lista de direcciones mac usadas con multicast.

mc_count Número de direcciones multicast usadas por el device de red.

Información específica de protocolos También incorpora información para appletalk y la gestión balanceada de device de red.

buffs Array de buffers usados por los protocolos de alto nivel como IPv4 o IPv6 para encolar las peticiones pendientes de enviar. Como máximo pueden existir DEV_NUMBUFFS en curso.

Función de inicialización *int (*init)(struct device *dev);*
Se llama al tiempo de inicialización del dispositivo de bajo nivel, solo es llamada una vez. Si el dispositivo retorna 0 el kernel considera que el device se inicializó correctamente.

Estadísticas Existen dos funciones para la generación de estadísticas, *net_device_stats* y *iw_statistics*, según el tipo de estadísticas del dispositivo ha de proporcionar una función para mostrar una u otra.

open *int (*open)(struct device *dev)*
Función que se llama cada vez que se inicializa un device de red, o un *ifconfip up*.

close *int (*close)(struct device *dev)*
Inversa de *open*, se llama cada vez que un *Network Device* se desactiva.

hard_start_xmit *int (*hard_start_xmit) (struct sk_buff *skb,struct device *dev)*
Cuando los protocolos de alto nivel desean transmitir un *sk_buff* llaman a *hard_start_xmit*.

hard_header *int (*hard_header)(struct sk_buff *skb,struct device *dev,unsigned short type,void *daddr,void *saddr,unsigned len)*
Es llamada para que el device de red añada la cabecera en *skb*, a partir de los datos que recibe como parámetros. El device de red ha de encargarse de la reserva del espacio necesario.

rebuild_header *int (*rebuild_header)(struct sk_buff *skb)*
Se llama cada vez que se pretende crear la cabecera para un paquete ARP.

hard_header_cache *int (*hard_header_cache)(struct dst_entry *dst, struct neighbour *neigh, struct hh_cache *hh)*
Con la intención de no construir cabeceras iguales cada vez que se pretende enviar un paquete, el kernel utiliza una cache de cabeceras. Si cuando se llama esta función, se crea una cabecera y se retorna 0, cada vez que se pretenda enviar una paquete al mismo destino no será necesario volver a crear otra cabecera, esta misma cabecera será reaprovechada. Esta función solo ha de retornar distinto de cero en caso que la cabecera no incorpore información variable como puede ser el tamaño del paquete. Por este motivo se usa en todas las tramas Ethernet excepto en 802.3 que incorpora el tamaño del paquete.

header_cache_update *void (*header_cache_update)(struct hh_cache *hh, struct device *dev, unsigned char * haddr)*
Cuando se utiliza la cache en las cabeceras, existe la posibilidad de modificar la dirección hardware, para notificar ese cambio se llama a esta función.

change_mtu *int (*change_mtu)(struct device *dev, int new_mtu)*
Tal y como su nombre indica esta función es llamada cada vez que se cambia la mtu del *Network Device*.

3.5.3. Diseño de Devices de Red

Existe un gran variedad de hardware para interconectar equipos informáticos, con la intención de unificar el interface de este hardware existen unos devices de red de bajo nivel, todos poseen un interface común para poder operar con los protocolos de alto nivel.

En los fuentes de Linux los devices de bajo nivel encargados de la gestión de redes se denominan *Network Devices*. En muchos puntos del kernel también se denomina *Network Interface*.

Los devices de red se encargan de enlazar los protocolos de alto nivel (IP,ARP,IPX), con el hardware específico usado como medio de transmisión. Se ha de rellenar una estructura y registrar las funciones en el Kernel. Antes de crear un device de red se ha de tener muy claro el funcionamiento de los *sk_buff* y sobre todo la estructura *device*. Poseen la misma funcionalidad que los NIC, *Network Interface Card drivers* de Windows NT, aunque en la versión 4.0 de NT se ha incorporado un nivel intermedio de conversión entre los protocolos de alto nivel y los NIC.

Al diseñar un nuevo device se ha de tener en cuenta que normalmente todos los devices se crean como los modules descritos en la sección 3.3.

A la hora de diseño se ha de tratar que el tiempo de servicio de interrupción sea mínimo, los dispositivos lentos han de realizar las lecturas y escrituras dentro de botton halfs o threads con la intención de evitar colapsar la máquina, se ha de recordar que mientras se sirve una interrupción nadie puede entrar al kernel.

3.5.3.1. Funcionamiento del Device de red

En esta sección se muestra como un device de red, en concreto una placa Ethernet, interacciona con los kernel de Linux a la hora de enviar un mensaje a una maquina remota.

El entorno de trabajo consiste en dos máquinas interconectadas con placas NE2000, la principal se llama *nether* y la secundaria *starfighter*. El driver de red se ha compilado como un module, de forma que es necesario cargarlo antes de operar con él.

Existen seis pases bien diferenciadas en la operación con devices de red:

Inicialización detección del hardware y registro en el kernel.

Activación preparación para recivir y enviar mensajes por la red.

Transmisión enviar un mensaje a una máquina remota.

Recepción recepción de una trama.

Desactivación dormir la placa para poderse desactivar o volverse activar.

Eliminación eliminar del kernel todos los recursos usados por el device de red.

Existen dos entradas en el *Proc Filesystem* ver sección 3.4, que muestran información relevante sobre el estado del kernel en relación con los devices de red.

El directorio */proc/net/dev* posee la lista de devices de red que se han registrado en el kernel de Linux, junto con las estadísticas de cada device. Después de botar *nether*, solo ha de poseer una entrada para el loopback *lo*. Con la intención de no mostrar demasiada información se ha suprimido las estadísticas de transmisión.

```
Inter-|   Receive
 face |bytes    packets errs drop fifo frame
    lo:    3048      44    0    0    0    0
```

El directorio */proc/net/arp* incorpora todas las direcciones arp que ha aprendido, así como su estado actual.

```
Address         HWtype  HWaddress           Flags Mask            Iface
e2              ether   00:00:21:86:72:67   C     *               eth0
e1              netrom  00:00:00:00:00:00   MP    *               *
```

Inicialización La primera operación ha de consistir en cargar los modules de NE2000 [10]. Esto se consigue con las siguientes instrucciones:

```
$nether>$ insmod 8390
$nether>$ insmod ne io=0x300 irq=10
```

Ahora el directorio */proc/net/dev* incorpora un nuevo device de red que corresponde a la placa de red NE2000.

```
Inter-|   Receive
 face |bytes    packets errs drop fifo frame
    lo:    3048      44    0    0    0    0
  eth0:       0       0    0    0    0    0
```

Cuando se inicializa el module se ha de acostumbra a realizar el siguiente proceso:

1. Detectar el hardware que ha de utilizar el *Network Device*, si consigue detectarlo ha de reservar los recursos usados (memoria, irq...).

2. Re-inicializar el hardware a un punto estable.

3. Rellenar la estructura *device*, son imprescindibles los campos: *open*, *close*, *hard_start_xmit*, *init mtu*, *flags*, *hard_header_len*, *type*. Aunque normalmente es necesario usar muchos más parámetros. Es conveniente revisar toda la estructura device [11] para observar que ha de ser rellenado.

4. Llamar a la función *register_netdev* pasándole como parámetro la estructura device que se ha rellenado. Esta función de encarga de registrar el device de red dentro del kernel.

[10] Esta operación puede realizarse automáticamente con el kerneld tal y como se explica en el capitulo 3.3
[11] En 3.5.2 se comentan todos los parámetros necesarios en la versión *Linux 2.1.40*

Activación Se produce cuando se asigna una dirección de algún protocolo superior al device, en nuestro caso trabajamos con IPv4. Esta operación se realiza mediante el comando ifconfig.

```
$nether>$ ifconfig eth0 nether
```

Al ejecutar el comando anterior se llama a la función *open* de la estructura *device*. Si se retorna distinto un valor negativo la activación será cancelada.

La figura 3.3 muestra como se suceden las llamadas a sistema para realizar un ping entre nether y starfighter.

Transmisión y Recepción Una vez se ha realizado la activación ya se puede proceder a enviar información entre dos nodos.

```
$nether>$ ping starfighter
```

Al realizar un ping contra starfighter, se ha re recordar que no posee la dirección de destino, por ese motivo la ha de aprender mediante el protocolo ARP. Se produce la siguiente secuencia de funciones:

1. Se llama la función *hard_header* para construir la cabecera del paquete ARP query a enviar.

2. Se envía un paquete de tamaño 42 bytes (ARP query) mediante la función *hard_start_xmit*.

3. Mientras que starfighter procesa el arp reqply, nether llamar a *hard_header_cache*, para ver si puede realizar un cache de la cabecera.

4. Llega un paquete de starfighter y se propaga **sin la cabecera de bajo nivel** al nivel superior mediante la función *netif_rx*. Se ha de crear un *skb* con el protocolo que ha llegado y la trama recibida.

5. Al procesar el arp reply se llama a *header_cache_update* para que actualice la dirección hardware de starfighter.

6. Ahora se llama enviar la información deseada mediante la función *hard_start_xmit*. No es necesario regenerar la cabecera de la trama pues está cacheada.

7. Se genera otra interrupción en la placa de red y procesa el mensaje que reenvía al nivel superior mediante *netif_rx*

Eliminación Cuando se descarga el module se ha llamar a *unregister_netdev* y después liberar todos los recursos adquiridos. Para poder eliminar un module el device de red ha de estar down, es decir en estado de desactivación o justo después de inicialización.

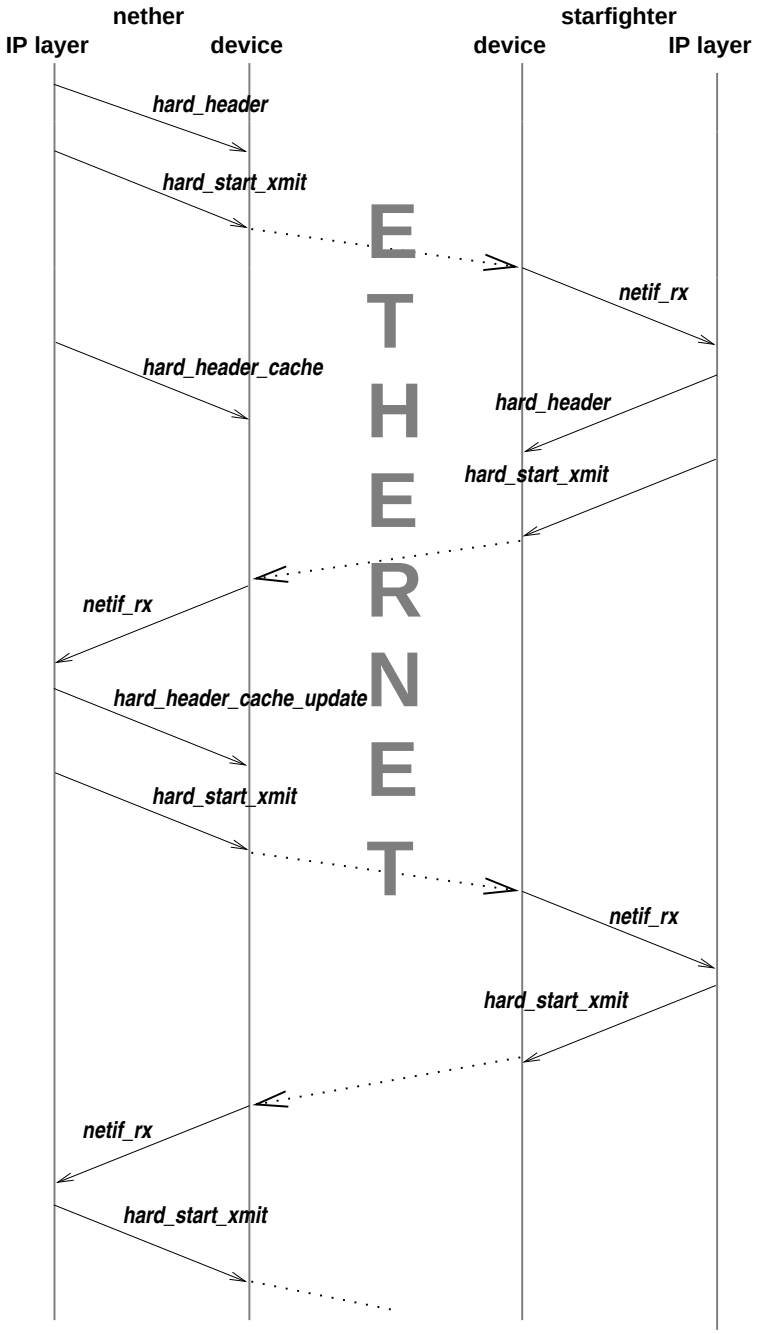

Figura 3.3.: Secuencia de llamadas al device de red para realizar un ping

3.6. El subsistema SCSI

Esta sección pretende dar una visión tanto explicativa como de punto de referencia sobre como trabaja el subsistema de SCSI a nivel de núcleo del sistema operativo.

Se irán comentando los diferentes niveles o módulos lógicos de los que se compone el sistema total, empezando por la parte de más bajo nivel como son las controladoras, siguiendo con el nivel intermedio o genérico y acabando por el nivel superior o de control de dispositivos.

Una vez vistas las estructuras y definiciones de funciones con las que cuentan cada uno de los niveles, se detallará cual es el procedimiento de la inicialización de todo el subsistema, tanto en modo estático como con *modules*.

3.6.1. Introducción

El sistema operativo Linux, es uno de los sistemas operativos que soporta más variedad de dispositivos y tarjetas SCSI del mercado. Ello es así, debido a que hay mucha gente usándolo, y por lo tanto existe una gran variedad de hardware con el que se trabaja. Esto ha propiciado que la gente que se encontraba con un hardware no soportado aun, lo construyera o contactase con alguien para que lo hiciera.

Con todo, la organización estructural del subsistema SCSI, esta bien diseñada y está estructurada modularmente al igual que muchas otras partes del kernel.

El código de SCSI, esta subdividido en varias capas lógicas:

Nivel Bajo Este nivel lo forman los drivers específicos de los adaptadores SCSI. Cada driver debe proporcionar una serie de funciones comunes para que el nivel medio pueda trabajar de la misma manera con cualquier adaptador de bajo nivel.

Nivel Medio El nivel medio as quizás la más importante, ya que es la que proporciona un acceso homogéneo para los distintos dispositivos del bus. Esta es la capa que implementa la abstracción común de dispositivos SCSI, lo cual es la idea inicial de diseño del bus.

Nivel Alto La capa de mas alto nivel la forman los drivers de dispositivos SCSI que se encuentran en el bus. Aquí se encuentran los controladores de disco, cinta,etc. . . Estos dispositivos usan las funcionalidades que les da el nivel medio, para poder tratar con el protocolo SCSI a su dispositivo.

Debido a esta variedad de hardware SCSI soportado, el subsistema SCSI es uno de las partes internas de kernel más grandes, ya que el código genérico está preparado para un gran numero de excepciones y *bugs* de las tarjetas actuales. Por otro lado, la flexibilidad que nos dan los modules, nos permite poder cargar y descargar cualquier módulo de alto o bajo nivel, la cual cosa reduce sensiblemente el tamaño del kernel estático. El nivel medio, no es tratado como un module, ya que se necesitan algunas funcionalidades como las de inicialización y transmisión de comandos para poder inicializar los modules que se

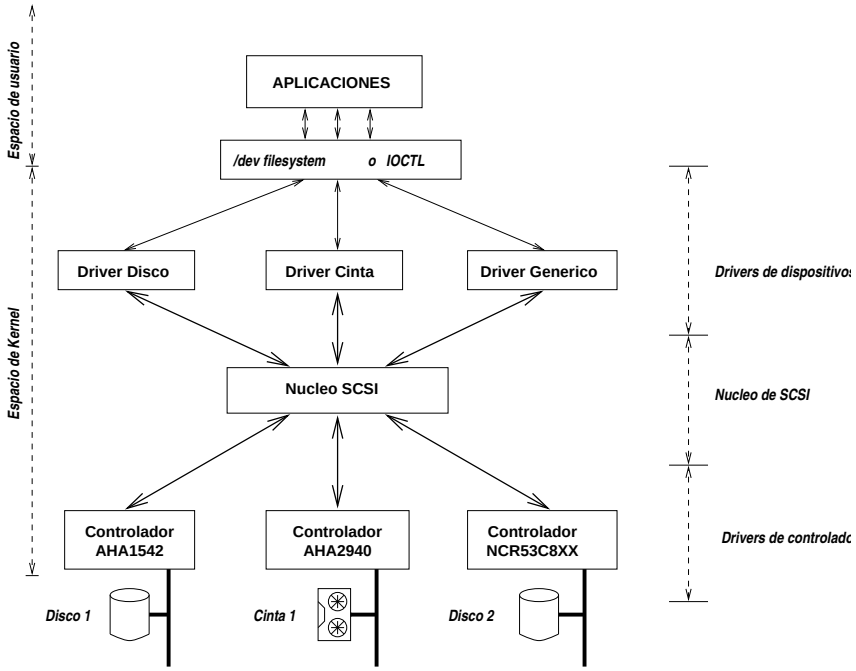

Figura 3.4.: Estructura del subsistema SCSI de Linux

vayan cargando. A todo eso, es posible que en un futuro también se pueda tratar como otro module si se modifica el código.

A continuación se van a explicar con mas detalle los diferentes niveles anteriormente citados, así como cuales son las definiciones y interacciones entre ellos.

3.6.2. Drivers de controladoras

Este es el denominado Bajo nivel, que como ya se ha dicho tratará los detalles específicos de la controladora para poder dar una interfaz de funciones concreta y definida. Así pues, cada driver de controladora SCSI deberá implementarse utilizando y creando las estructuras y funciones que el nivel SCSI define para los *hosts*(controladoras) del sistema. El subsistema SCSI, pues, define las dos siguientes estructuras, que serán comunes para todo el bajo nivel:

Scsi Host Template Es la definición de una "plantilla" de los datos mínimos que se necesitan para la interacción con el nivel medio. Habrá una plantilla diferente para cada tipo de controladora diferente.

Scsi Host Esta estructura contiene todos los datos posibles sobre las controladora física. Cada *Scsi Host* pertenecerá al mismo tiempo a una determinada plantilla, según el tipo de hardware que sea.

Así pues, si por ejemplo tenemos dos controladoras aha1542 y una aha2940 en el sistema, entonces se crearan las siguientes estructuras:

- Un *Scsi Host Template* para el tipo de controladora aha1542
- Un *Scsi Host Template* para el tipo de controladora aha2940
- Dos *Scsi Host* para las aha1542 (uno para cada tarjeta). Los valores que variaran serán los de las interrupciones, dma, y otros.
- Un *Scsi Host* para la aha2940.

Por lo tanto, para una cada controladora existente, se deberá crear una estructura *Scsi Host* y una *Scsi Host Template*. En caso que ya hubiera la plantilla porque hay otras controladoras iguales, entonces no se volverá a crear.

Entonces veamos, que es lo que tiene que implementar el nivel bajo, para completar todas estas estructuras.

3.6.2.1. Scsi Host

Esta estructura solo tiene los datos propios referentes a una controladora física del sistema, pero además tiene una referencia que indica a que tipo de plantilla o *Scsi Host Template* pertenece. Por lo tanto, los datos y funciones genéricas que la identifican se pueden conseguir mediante su plantilla.

Estos son los datos más relevantes de la estructura:

Colas Tiene definidas varias colas, como por ejemplo la cola de espera (para los comandos), o la cola de hosts para la exclusión mutua de cosas como el uso del mismo canal de *DMA*.

Valores locales Son aquellos valores propios de la tarjeta que la identifican de las otras del sistema. Por ejemplo, aquí hay incluidos elementos como el *Id* y *LUN* SCSI, dirección base de E/S, irq, canal de *DMA*, etc...

Copias de la plantilla Hay algunos valores que tiene la plantilla del host, que puede interesar en un cierto momento cambiar para un *host* determinado, por ejemplo reducir el numero de comandos encolables durante un cierto tiempo. Es por ello, que hay algunos valores que aunque estén en la plantilla, son copiados también en esta estructura para cambiarlos si se presta.

Aunque la esta estructura *Scsi Host* sea la que hace referencia a la controladora física, la que tiene más información de como trabaja es la estructura *Scsi Host Template*, ya que contiene todas las funciones y generalidades del tipo concreto.

3.6.2.2. Scsi Host Template

Los parámetros mas relevantes de esta estructura, y que el driver debe proveer son los siguientes:

Proc Las funciones que le permitan gestionar una entrada en el *proc filesystem* (ver sección 3.4).

Datos específicos Como son el nombre o tipo de la controladora, cuantos comandos encolados soporta, cuantos comandos por *lun* permite, etc...

Funciones de activación Las funciones que permiten la detección y la desactivación de la tarjeta. La función de detección `detect()` es necesaria y muy importante.

Funciones de operación Son las que le permitirán al nivel superior tratar con la controladora. Estas funciones son `command`, `queuecommand`, `abort` y `reset`.

A continuación se detalla cuales son las características que deben tener las funciones principales que deben ser definidas por el driver de bajo nivel:

`detect()` Esta función es la que se encargará de detectar la tarjeta, deberá registrar el host en el subsistema Scsi (`scsi\_register()`), inicializarlo, rellenar los parámetros adecuados de dirección base, irq, dma... en la estructura de *Host*, preparar la memoria que precise, reservar una interrupción y un canal de dma en el sistema (en caso de ser necesario) y instalar el gestor de interrupciones. Una vez finalizada la rutina, la tarjeta estará inicializada, sus estructuras rellenadas y preparada para funcionar.

`command()` Esta función recibirá un comando Scsi (*Scsi_Cmnd*) y deberá ejecutar-lo en su controladora de la manera apropiada. También deberá ejecutar la función `scsi\_done()` al finalizar el comando.

`queuecommand()` Esta función tiene la misma misión que la anterior, pero con la diferencia que el comando será encolado en la lista de comandos de la controladora, con lo cual se ejecutará según su prioridad y situación. La rutina `scsi\_done()` pasada como parámetro, será guardada en el campo del mismo nombre en la estructura `Scsi\_Cmnd` para que se ejecute cuando el comando haya finalizado (en la rutina de interrupción).

`abort()` Esta rutina debe encargarse de acabar o abortar el comando recibido como parámetro, que es un comando Scsi anteriormente encolado a su controladora.

`reset()` Debe ser capaz de causar un *reset* a la controladora o al bus SCSI.

`intr_handler()` Esta función sera el controlador de interrupciones de la placa, y lo instalará la función `detect()`. Se deberá encargar de la gestión de las interrupciones que genere el *host* y actuar dependiendo de la causa de ellas. A todo ello, las cosas que debe hacer cuando se finaliza un comando son:

- Liberar memoria si hace falta.
- Computar el código de error (en caso de haberlo).

- Copiar el *sense_buffer* a la estructura del comando Scsi, si este lo requiere.
- Llamar a la función `scsi\_done()` especificada en el comando.

`module` Las funciones `init\_module()` y `cleanup\_module()` en caso de trabajar con *modules* solo deberán registrar o "desregistrar" al host llamando a las funciones de scsi: `scsi\_register\_host` con el parámetro de *Host Adapter*, o a la función `scsi\_unregister\_host`.

Todas estas funcionalidades, son las que se deben implementar correctamente si se quiere crear un nuevo driver SCSI en el sistema. Todo ello, sabiendo manejar las demás estructuras del sistema, como los comandos Scsi y las funciones genéricas como pueden ser las de registro de dispositivos del sistema,etc...

Veamos, que proceso sigue el kernel para inicializar el bajo nivel, utilizando las estructuras de las controladoras explicadas anteriormente.

3.6.3. Inicialización del nivel bajo

Hay ciertas funciones y variables globales del sistema que provee el kernel para el control de todos los drivers de controladoras.

Hay dos variables globales que contienen las listas de hosts:

`scsi_hosts` Contiene la lista de plantillas de controladoras (*Scsi Host Template*) existentes en el sistema.

`scsi_hostlist` Contiene la lista de hosts físicos (*Scsi Host*) existentes y activos del sistema.

Estas variables y sus estructuras, tomando como ejemplo el anteriormente comentado, compuesto por dos tarjetas aha1542 y una aha2940, estarían modeladas en la figura 3.5

Por otro lado, las funciones genéricas mas importantes son las siguientes:

`scsi_register` Se utiliza para registrar un nuevo *host* a la lista del sistema.

`scsi_unregister` Es utilizada para "desregistrar" un adaptador anteriormente registrado.

Utilizando estas funciones y variables, la función básica de inicialización y detección, llamada cuando bota el sistema, es `scsi\_init`. Los pasos más importantes que realiza son los siguientes:

- Detecta a todos los *host*. Lo hace llamando a todas las funciones `detect()` de los templates.
- Registra a todos los hosts que se han detectado, mediante `scsi\_register`.
- Les crea el *proc* si el driver lo soporta.
- Registrara a todos los drivers de dispositivos de alto nivel (disco, cinta...)

Cabe notar que la función que realiza estas operaciones para los hosts que se cargan como modules, es `scsi\_register\_host`.

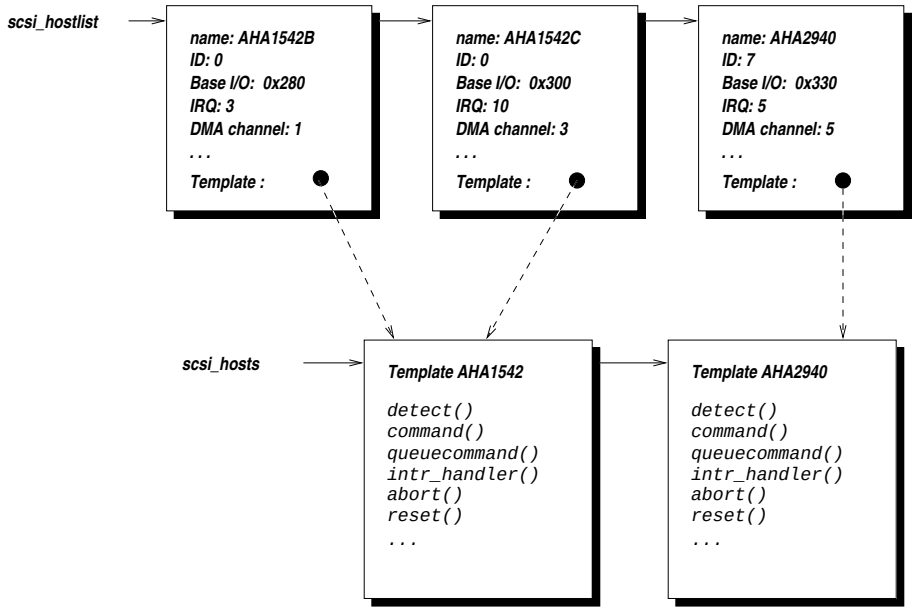

Figura 3.5.: Ejemplo de configuración de listas

3.6.4. El núcleo del sistema

Por núcleo del sistema o parte principal, se entiende la parte genérica propiamente dicho del subsistema SCSI. Esta parte corresponde al nivel medio de la estructura comentada en la sección 3.6.1

Esta parte tiene todas las definiciones genéricas de comandos, funciones para enviarlos o manejarlos, y todas las definiciones de funciones para tratar con los subsistemas de bajo y alto nivel.

Las estructuras y funciones más importantes con las que trata este nivel son:

- Estructura de comando Scsi
- Rutinas de inicialización, detección y identificación de hosts
- Rutinas de inicialización de drivers de dispositivos (de alto nivel)
- Rutinas para el manejo y envío de comandos SCSI.

A continuación de dará una visión más detallada del la definición que hace el núcleo de los comandos SCSI, así como su manejo y ejecución. La parte de inicialización del sistema, aunque se encargue el núcleo de SCSI, se verá al final del capítulo cuando se hayan visto con más detenimiento los niveles bajo y alto, debido a que esta inicialización se basa en muchos conceptos de estos niveles.

3.6.4.1. Estructura del comando SCSI

La estructura que define el kernel sobre el comando SCSI, es única y siempre se debe utilizar la misma. En ella hay definidos todos los campos necesarios para poder saber como gestionar el comando, a quien enviarlo, el comando de bus al que hace referencia, etc...

La tabla 3.6.4.1 muestra la forma y los datos que contiene esa estructura.

Campo	Descripción
host	Puntero al host al que pertenece el comando
target,lun,channel	Identificadores del host
cmd_len	Longitud del comando específico SCSI
cmd	Es el comando SCSI (array de bytes)
use_sg,sglist_len	Parámetros que indican si es un comando de *scatter-gather*
abort_reason	La razón, por la cual el nivel medio ha pedido abortar este comando (si la hay)
buffer,bufflen	Longitud y puntero a los datos del comando
sense_buffer	Array de bytes reservado para copiar el resultado de un comando `REQUEST SENSE`, en caso de que haya habido un error
"varios"	Campos para el manejo de los timeouts y reintento del comando
scsi_done()	Función que ejecutara el nivel bajo al finalizar el comando
done()	Función que ejecutará el nivel medio al acabarse el comando
host_scribble	Puntero genérico que puede utilizar el nivel bajo según disponga
result	Campo donde el nivel bajo pondrá el resultado del comando al finalizarse

Cuadro 3.2.: Estructura del comando scsi (*Scsi_Cmnd*)

Como puede observarse, es una estructura bastante grande. Esto es debido a que es lo mas genérica posible, esta pensada para ser usada en conjunción con el nivel bajo y además tiene información adicional al comando en sí, como es el control de *timeouts* o de reintentos de comandos fallidos.

Hay algunos de estos campos que merecen una más extensa explicación.

Los campos indispensables para cualquier transmisión de información, evidentemente serán los que nos definirán al *target* al cual queremos seleccionar (`lun`,`id`...), el comando Scsi (`cmd`) y la longitud y puntero a la zona de datos (`buffer` y `bufflen`).

Existen unos campos, no especificados en la tabla, llamados `request\_buffer` y `request\_bufflen` que son copias de los campos `buffer` y `bufflen` que se hacen en ciertos momentos en el transcurso de la ejecución del comando, por ejemplo en la retransmisión de comandos. También sirven de copia para el `sense\_buffer` cuando se esta realizando un comando de `REQUEST SENSE`.

Dos campos primordiales en el manejo de los comandos, son las dos funciones que se ejecutan al finalizar el comando al que hacen referencia (`scsi\_done()` y `done()`). El procedimiento es el siguiente:

1. El nivel superior utiliza la función `scsi\_do\_cmd()` para ejecutar un comando, pasando como parámetro una función a ejecutar cuando su comando finalice. Esta función es la que se colocará en el campo `done`

2. La función `scsi\_do\_cmd()` acabará en `internal\_cmnd()`, donde apuntará y actualizará las variables de los *time-outs*. Después, esta llamará a la función de bajo nivel `queuecommand()`, para encolar un mensaje en la controladora, pasando como parámetro la función que debe ejecutarse al acabar el comando. Esta función se deberá ejecutar antes que la del nivel superior.

3. La función `queuecommand()` pondrá en el campo `scsi\_done`, el parámetro que le ha pasado el nivel medio (`scsi\_done()`). Esta función es la función genérica de tratamiento de finalización correcta o incorrecta de comandos por parte del nivel medio.

4. El acabarse el comando, el nivel bajo (la rutina de interrupción programada) llamará a la función que tenga en el campo de `scsi\_done()` del comando que ha finalizado. En este caso llamará a la función de nivel medio `scsi\_done()`.

5. La función `scsi\_done()` se encargará de ver si el comando se ha completado con éxito y en tal caso, llamar a la función que haya en el campo de `done`, que en este caso es la función de nivel alto. En caso que haya habido un error se volverá a reintentar el comando, o se abortará...

Como se puede deducir, la función `scsi\_do\_cmd()` juega un papel fundamental en el sistema porque es la función que tendrán que llamar todos los dispositivos que quieran enviar un mensaje. Ya se ha visto que además de enviar los comandos, se encarga de manejar los posibles *time-outs* y retransmisión de comandos. Además de eso, también previene posibles exclusiones mutuas entre diferentes drivers de dispositivos (alto nivel) atacando a la misma controladora.

3.6.5. Drivers de dispositivos

Este nivel de SCSI, es el mas alto, la cual cosa significa que utiliza las funcionalidades que le proporciona el nivel medio o de núcleo.

En este nivel se construyen los diferentes drivers para los diferentes tipos de dispositivos que se tiene en el bus SCSI, como por ejemplo controladores de disco (*sd*), de cinta (*st*) o genéricos (*sg*).

Las cosas básicas que debe definir un driver en este nivel, son las funciones de inicialización, detección y manejo de los diferentes dispositivos, representados por una estructura común: `Scsi\_Device`.

3.6.5.1. Scsi_Device

Hay una definición igualmente genérica que los drivers de dispositivos deben emplear para su funcionamiento y inicialización. Esta estructura es la denominada `Scsi\_Device` y la filosofía es la misma que en la estructura para los hosts, es decir, agrupar toda la información necesaria para el manejo, detección y identificación del dispositivo al que referencia. Esta estructura consta principalmente de los siguientes campos:

Campo	Descripción
id, lun, channel	Identificadores SCSI del dispositivo
attached	Numero de drivers asociados
host	Host por el que se accede al bus
wait_queue	Colas de espera para los drivers que usan el dispositivo
device_queue	Lista de comandos encolados al dispositivo
tipo, nombre...	Parámetros específicos del dispositivo
"parámetros"	Parámetros para definir algunas de las opciones de SCSI, como si soporta *tagged-queuing*, multiples *LUNS* ...

Cuadro 3.3.: Estructura del dispositivo scsi (*Scsi_Device*)

La lista de dispositivos activos en el sistema se encuentra en la variable global de kernel `scsi\_devices`, donde hay encadenadas estas estructuras debidamente cumplimentadas.

Tal y como se deduce por los campos de la estructura, los drivers dispositivos compartirán estas estructuras para acceder a los dispositivos SCSI del bus, bloqueándose para la exclusión mutua en caso de colisión o colas de comandos llenas. Todo será manejado por el núcleo de SCSI

El nivel medio de SCSI, proporciona dos funciones genéricas encaradas a los drivers de alto nivel, para poder pedir genéricamente estructuras libres de comandos SCSI de la controladora asociada. Son las siguientes:

request_queueable() Intenta retornar una estructura del tipo `Scsi\_Cmnd` válida para el dispositivo. Tiene en cuenta la estructura interna de la controladora, así como la longitud de la cola de comandos permitida y la exclusión con otros drivers de dispositivos que la usen. En caso de que no pueda retornar un comando libre o válido, retorna `NULL`.

allocate_device() Tiene el mismo comportamiento que la anterior función, con la excepción que se le puede indicar mediante un parámetro, que se quede bloqueado esperando hasta que haya un comando libre.

3.6.5.2. Definiciones de los drivers

Cada driver de dispositivo de este nivel, deberá completar una estructura común indicando las cosas necesarias para interactuar en el sistema. Esta estructura es del tipo

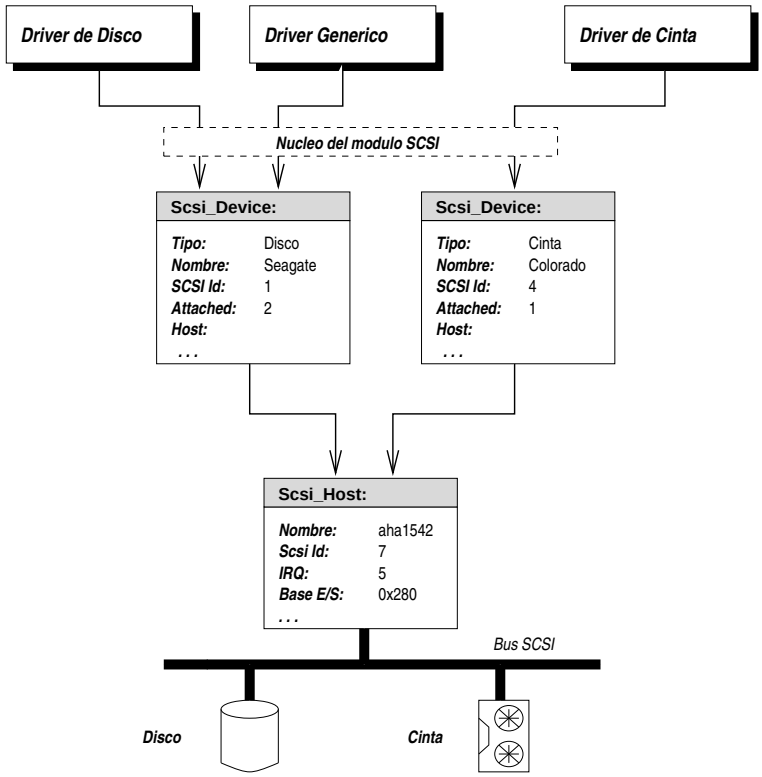

Figura 3.6.: Sistema de compartición de dispositivos proporcionado por el núcleo del sistema SCSI, a través de las estructuras *Scsi_Device*.

`Scsi\_Device\_Template` y tiene una definición similar a la que se veía en la estructura `Scsi\_Host\_Template` en la sección 3.6.2.2. Los campos más importantes son:

Identificadores dispositivo Campos de nombre, tipo y otros que identifican al dispositivo que manejará el driver.

Valores propios Como *major_number*, numero de drivers detectados y adjuntos, y otros valores para el control interno del driver.

detect() Función que deberá retornar 1 en caso que detecte el *scsi_device*, como un tipo de dispositivo para su control. Sino, debe retornar el valor 0.

init() Deberá encargarse de hacer las inicializaciones necesarias en función de los dispositivos que haya detectado. Normalmente se usa para la reserva de memoria.

attach() Se encargará de realmente hacer la inclusión del dispositivo bajo el control del driver.

finnish() Se llamará para cada dispositivo, una vez se haya hecho un `attach()`. Sirve para hacer algo más una vez ya inicializado el dispositivo.

detach() Deberá hacer las acciones pertinentes para desvincular el driver del dispositivo.

Una vez que el driver ha definido estos parámetros, más los que necesite internamente, el subsistema SCSI se encargará de hacer la secuencia de llamadas necesarias para que estas funciones sean llamadas con el orden correcto.

3.6.6. Inicialización del sistema

La inicialización del sistema SCSI, está a cargo núcleo , pero se hace utilizando las funciones y estructuras que los niveles bajo y alto, que han debido implementar correctamente, tal y como se ha explicado en las secciones 3.6.2 y 3.6.5.

Hay dos diferencias fundamentales en la inicialización, según se haga en tiempo de *boot* o mediante el sistema de carga de *modules*.

Debido a que la diferencia es básicamente el nombre de las funciones que se usan, la explicación se basará en la inicialización en tiempo de boot ya que internamente, el mecanismo será el mismo cuando se hace con el sistema de *modules*

La rutina principal de inicialización del sistema es `scsi\_dev\_init()`, la cual primero efectuará la inicialización de los hosts, y después procederá a la de los drivers de dispositivos, según los hosts que haya encontrado.

3.6.6.1. Inicialización de hosts

La rutina principal que controla en inicio del nivel bajo es `scsi\_init()` y su funcionamiento simplificado es como sigue:

- Recupera la lista de todos las plantillas de hosts posibles, que se han configurado al compilar el kernel (`builtin\_scsi\_hosts`).

- Para cada uno de la lista, les llama la función `detect()`, guardando el resultado en el campo present de la plantilla. Esta operación causa que cada tipo de host intente detectar y inicializar tantas controladoras de su tipo como sea posible. Un resultado positivo indica que se ha encontrado algún numero de controladoras de ese tipo.

- Para cada una de las plantillas detectadas positivamente, se registra el host en el sistema SCSI, mediante la función `scsi\_register()`.

- Para cada uno de los hosts de la lista `scsi\_hostlist`, creada al ir registrando hosts en el paso anterior, imprimir un mensaje indicando nombre y tipo.

- Después, se registran estáticamente todos los controladores de alto nivel. Se hace aquí por razones de facilidad, ya que de esta manera, al ir detectando dispositivos como se verá más adelante, se podrá ir interactuando con estos drivers de dispositivos, sin necesidad de hacer varias veces la detección en el bus.

3.6.6.2. Inicialización de drivers de dispositivos

Una vez ya se tienen todos los hosts del sistema detectados y inicializados, la rutina de inicialización se encargará de detectar todos los dispositivos existentes el todos los buses SCSI detectados, y usarlos como parámetros para las rutinas que los drivers de alto nivel habrán definido en sus respectivas plantillas, ya registradas en la inicialización de los hosts.

La inicialización se basa en la función `scan\_scsis()` que en combinación con la función con la función `scan\_scsi\_single()` serán las encargadas de detectar mediante comandos `TEST UNIT READY` los dispositivos de los buses.

En caso de obtener respuesta a estos comandos, se encargarán de enviar un comando adicional (`INQUIRY`) para saber de que tipo de dispositivo se trata, y poder pasar esa información (función `detect()`) a los drivers de alto nivel registrados. Un valor retornado superior a 0, por parte de la función `detect()` indica que el driver reconoce el dispositivo como suyo.

Una vez hecho esto, el núcleo de scsi llamará a la función `init()`, `attach()` y `finnish()` de los drivers pasándoles los dispositivos detectados, y por el orden descrito.

De esta manera es como el subsistema de SCSI logra inicializar los hosts y los drivers de dispositivos con una sola búsqueda en el bus.

3.6.6.3. Modules

La inicialización del sistema utilizando modules utiliza las mismas funciones internas y filosofía de detección, pero varían los puntos de entrada. Con lo cual, los modules deberán utilizar las siguientes funciones:

scsi_register_module() Esta función es la utilizada por los *modules* para añadir un driver de controladora. También existe en equivalente para "desregistrarla" (`scsi\_unregister\_module()`).

scsi_register_device_module() Esta es la función que deben usar los *modules* para registrar un nuevo driver de alto nivel. Esta función internamente hará prácticamente lo mismo que la anteriormente explicada `scsi\_dev\_init()`.

scsi_unregister_device() Es la inversa a la anterior, y sirve a los *modules* para "desregistrar" los drivers de dispositivos.

Capítulo 4 ❑

Puerto Paralelo

4.1. Introducción

El puerto paralelo es muy popular, aparece en todos los PC y permite un sistema de interconexión entre ordenador y mundo exterior con multitud de dispositivos adaptables.

En un comienzo apareció como un sistema de conectar impresoras de forma alternativa al lento puerto serie. Poco a poco aparecieron más periféricos, actualmente existen CDROM, unidades magneto-ópticas, placas Ethernet ...

También se utiliza en aplicaciones específicas como sistema de entrada y salida de datos, para pequeños proyectos que necesitan un intercambio de información entre un ordenador y un periférico.

Aunque su popularidad se debe al ser introducido en todos los PC, también aparece en Machintosh, Amiga, Atari, IBM mainframes, algunas maquinas SUN...

Como consecuencia de su popularidad el puerto paralelo ha ido incorporando nuevas características, actualmente varios modelos de interconexión y no todos lo chips se comportan de igual forma. En este capítulo se trata de aclarar cuales son las formas de operación del puerto paralelo, así de las notas que se han de considerar al diseñar aplicaciones con el puerto paralelo.

El puerto paralelo opera con múltiples señales TTL de forma simultanea, de forma que es capaz de transmitir varios bits de forma simultanea, 4 u 8 bits dependiendo del modo de operación.

El puerto original poseía 8 lineas de salida, 5 entradas y 4 lineas bidireccionales, en los ordenadores actuales, las 8 lineas de salida también pueden usarse de entrada.

Diseñado para poder operar con impresoras la nomenclatura de las señales corresponden a este uso (PaperEnd, AutoLineFeed ...). No obstante con la aparición de nuevos modos también han aparecido nuevas nomenclaturas que se explican al mismo tiempo que se explica cada modo.

A medida que el diseño del PC evolucionaba se han incorporado nuevas características al puerto paralelo. Siempre parten de un modo compatible desde el que se puede "saltar" a nuevos modos con otras características.

Existen 4 modos principales de operación de los puertos paralelos, aunque actualmente están apareciendo chips que permiten conmutar entre modos.

SPP **Standard Parallel Port** o Modo **Original** del puerto paralelo, todos los chips del mercado soportan este modo de operación, y normalmente este es el modo en el que se encuentra el puerto paralelo despues de botar la máquina. El diseño original se basa en el conector *centronics* usado en algunas impresoras. El modo SPP permite transferir 8bits hacia el periférico, usando un protocolo similar al usado en el conector *centronics*. Este modo permite una comunicación bidireccional mediante *Nibbles*. Aunque cada nibble solo incorpora 4bits y es un sistema lento de transferencia se ha convertido en un sistema ampliamente utilizado.

PS2 La primera mejora que apareció sobre el puerto paralelo consistió en un diseño de IBM que permitía que los 8bits de salida del SPP pudieran operar como lineas bidireccionales.

EPP **Enhanced Parallel Port**, fue diseñado por Intel, Zenith y Xircom con la intención de facilitar el diseño de placas Ethernet conectadas al puerto paralelo. Al igual que en el modo PS2 las lineas de datos son bidireccionales, no obstante opera a la velocidad del bus AT. Puede realizar una operación de lectura o escritura en solo 1us, incorporando negociación. El modo EPP permite cambiar de dirección de forma rápida, de forma que es muy eficiente en dispositivos que transfieren en ambas direcciones.

ECP **Extended Capabilities Port** esta propuesto por Hewlett Packard y Microsoft. Al igual que el modo EPP opera a la velocidad del bus AT, pero a diferencia del EPP puede operar con DMA y compresión de datos. Es útil para dispositivos que necesitan transferir bloques de información. La especificación del ECP incorpora modos de emulación de los modos PS2 y EPP.

4.2. Conceptos Básicos

La especificación original de IBM define en que direcciones pueden estar mapeados los puertos paralelos. Aunque han existido múltiples mejoras sobre la especificación original, desde el principio se han respetado estas tres direcciones, **0x278**, **0x378** y **0x3BC**.

De igual manera se especificó en que orden se tienen que detectar. Primero 0x3BC, despues 0x378 y por último 0x278. Al botar el ordenador la BIOS almacena en el intervalo 0x408-0x410 los puertos que ha detectado correctamente [1]. Esta secuencia es conocida en MSDOS como LPT1, LPT2, LPT3.

Las máquinas que soportan PnP se comportan de diferente forma, pues permiten que el software pueda obtener información adicional de la placa, explicar como funciona el PnP de ISA sale del enfoque de esta memoria, por este motivo si se esta interesado es aconsejable leer [Mic94].

4.2.1. Las señales

El modo SPP, el puerto paralelo original, se gestiona con tres registros, **data**, **control** y **status**, cada uno de estos registros dispone de 8 bits. Mediante estos tres registros se puede gestionar el conector del puerto paralelo que dispone de 25 lineas. El conector más utilizado es un DB25, aunque existen otras dos especificaciones. En la tabla 4.1 se muestran cada una de las lineas, así como al registro al que están asociadas.

Se ha de recordar que algunas lineas del conector DB25 poseen un valor invertido al correspondiente del registro asociado, por ejemplo la señal *Busy* del conector está asociada al séptimo bit del registro Status, pero mientras el registro es cierto, el valor en el conector es falso.

La mayoría de los nombres se heredaron del estándar centronics, y se han mantenido incluso con los nuevos modos. Todos los modos existentes permiten operar en SPP, por este motivo todos disponen de los tres registros básicos para poder controlar el conector DB25, aunque incorporan nuevos registros para añadir más funcionalidades.

La tabla 4.2 muestra todos los registros existentes, para los modos SPP, PS2, EPP y ECP. Existen 10 registros pero tal y como se puede observar en la tabla 4.3 varios registros coinciden en la misma dirección, aunque poseen un significado distinto según el modo en el que operan.

4.2.2. Nomenclatura

Durante todo el capítulo de puerto paralelo se usa una misma nomenclatura con tal de facilitar la comprensión. La nomenclatura utilizada está inspirada en la codificación C. La tabla 4.3 muestra las funciones utilizadas para acceder a cada registro. Se ha de observar que *base* corresponde a la dirección base del registro del puerto paralelo.

[1] Reserva dos bytes por cada puerto, existiendo un máximo de cuatro puertos posibles a detectar

Pin	Señal	Registro	Bit	Invertida
1	nStrobe	Control	0	Si
2	d0	Data	0	No
3	d1	Data	1	No
4	d2	Data	2	No
5	d3	Data	3	No
6	d4	Data	4	No
7	d5	Data	5	No
8	d6	Data	6	No
9	d7	Data	7	No
10	nAck	Status	6	No
11	Busy	Status	7	Si
12	PError o PaperEnd	Status	5	No
13	Select	Status	4	No
14	nAutoLF	Control	1	Si
15	nFault	Status	3	No
16	nInit	Contol	2	No
17	nSelectIn	Control	3	Si
18	Gnd (nStrobe,d0)			
19	Gnd (d1,d2)			
20	Gnd (d3,d4)			
21	Gnd (d5,d6)			
22	Gnd (d7,nAck)			
23	Gnd (nSelectIn)			
24	Gnd (Busy)			
25	Gnd (nInit)			

Cuadro 4.1.: Señales del DB25

Registro	D7	D6	D5	D4	D3	D2	D1	D0
data	d7	d6	d5	d4	d3	d2	d1	d0
Afifo	addr	a6	a5	a4	a3	a2	a1	a0
status	nBusy	nAck	PError	Select	nFault	xx	pirq	timeout
control	xx	xx	Direction	ackIntEn	SelectIn	nInit	autofd	strobe
Cfifo	d7	d6	d5	d4	d3	d2	d1	d0
Dfifo	d7	d6	d5	d4	d3	d2	d1	d0
Tfifo	d7	d6	d5	d4	d3	d2	d1	d0
cnfgA	ISA	wS2	wS1	wS0	xx	nByteIn	fs1	fs2
cnfgB	compress	intrValue	irq2	irq1	irq0	dma2	dma1	dma0
ecr	mode2	mode1	mode0	nErrIntrEn	dmaEn	serviceIntr	full	empty

Cuadro 4.2.: Registros del puerto paralelo

Registro	offset	Modos	Lectura	Escritura
data	0x000	SPP PS2 EPP	r_dtr(base)	w_dtr(base)
Afifo	0x000	ECP	r_dtr(base)	w_dtr(base)
status	0x001	todos	r_str(base)	w_str(base)
control	0x002	todos	r_ctr(base)	w_ctr(base)
Cfifo	0x400	ECP	r_fifo(base)	w_fifo(base)
Dfifo	0x400	ECP	r_fifo(base)	w_fifo(base)
Tfifo	0x400	ECP	r_fifo(base)	w_fifo(base)
cnfgA	0x400	ECP	r_cnfgA(base)	w_cnfgA(base)
cnfgB	0x401	ECP	r_cnfgB(base)	w_cnfgB(base)
ecr	0x402	ECP	r_ecr(base)	w_ecr(base)

Cuadro 4.3.: Mapeo de los registros

4.2.3. Registro de datos

El registro *data* o registro de datos posee los 8 bits de datos, en el modo SPP solo son de salida. En PS2, EPP y ECP se puede utilizar bidireccionalmente. Se ha de tener cuidado que el estándar IEEE1284 [IEE95b] el cual numera el registro de datos de D1-D8 en vez de D0-D7 como se hace en el resto de especificaciones.

Tal y como muestra la tabla 4.3 el registro de datos está mapeado en la dirección base donde se encuentra el puerto paralelo, es decir en 0x3BC, 0x378 o 0x278.

4.2.4. Registro status

El registro de status es de solo lectura y posee el valor de cinco lineas de entrada. No obstante cuando opera en modo EPP el bit 0 del registro de status, *timeout*, opera en escritura y lectura [2].

BIT 0 Timeout. En el modo EPP cuando este bit es cierto significa que se ha producido un timeout en una operación de transferencia de datos, tanto lectura como escritura. Solo se usa en modo EPP y no aparece en el conector como el resto de bits del registro status.

BIT 1 No usado.

BIT 2 PIRQ. Usado en muy pocos puertos paralelos indica el estado de las interrupciones. Es cierto si se ha producido una interrupción. Que este bit funcione depende del chip, no del modo de operación, por este motivo es poco aconsejable utilizarlo, a no ser que se trate de una aplicación para un hardware altamente especializado.

[2]Ver el modo EPP en la sección 4.4 para obtener más información del bit *timeout*

BIT 3 nFault. También se conoce como nError, usado por la impresora para notificar que se ha producido un error. Puede provocar una interrupción dependiendo del valor de *nAckIntEn*.

BIT 4 SelectIn. Cierto cuando la impresora esta operacional, cierto mientras que se opera en ECP.

BIT 5 PError. También se conoce como PaperEnd y PaperEmpty. En modo SPP posee una valor cierto cuando la impresora se ha quedado sin papel. En ECP se usa como acknowledge a una petición de cambio de dirección.

BIT 6 nAck. Transición de cierto a falso cuando la impresora recibe un byte. El cambio de *nAck* puede provocar una interrupción, dependiendo de *nACkIntEn*.

BIT 7 Busy Falso cuando la impresora no puede aceptar más datos.

4.2.5. Registro de control

El registro *control* se encuentra con un offset de 0x002 respecto a la base del puerto paralelo. Por reset los bits 0 al 5 se inicializan a falso.

BIT 0 Strobe. Mediante la transición falso cierto el periférico puede leer el valor de los datos. Su valor invertido se muestra en el conector.

BIT 1 AutoFD. Originalmente indicaba que la impresora tenia que realizar un *LineFeed* despues de cada *Carriage Return*. Ahora también tiene otros usos en los modos EPP y ECP.

BIT 2 nInit. La transición cierto falso y el mantenimiento de falso durante más de 50us provoca un reset en la impresora, eliminando los datos del buffer.

BIT 3 Select. Cierto indica a la impresora que active la entrada de datos. Es falso la impresora no esta seleccionada.

BIT 4 ackIntEn Cuando es cierto permite que la transición de falso a cierto de *nAck* provoque una interrupción.

BIT 5 Direction. En el modo SPP este bit no se utiliza. En los otros modos indica la dirección de los datos. Falso .0 de *Output*, los datos pueden ser transferidos del ordenador al periférico, mientras que un valor de cierto, 1 de *Input*, permite leer datos del bus. En algunos chips este bit solo se puede cambiar mientras operan en modo PS2, es decir que para cambiar de dirección en ECP y EPP primero se ha de pasar a modo PS2, despues cambiar el valor de *Direction* y luego volver al modo ECP o EPP.

BIT 6 No usado.

BIT 7 No usado, excepto por unos pocos chips en los que este bit es un alias del bit 5 (*Direction*).

4.2.6. Detección

La BIOS detecta los puertos paralelos escribiendo 0xaa en cada puerto de datos y despues lee el valor. Si posee el valor 0Xaa considera que el puerto existe.

Este sistema de detección puede tener varios problemas si el puerto paralelo no opera en modo SPP [3], también pueden existir problemas si existe algún dispositivo conectado al puerto paralelo que "fuerza" que las señales estén bajas.

Con la intención de evitar posibles problemas se puede usar el siguiente sistema de detección:

```
int SPP\_supported(int base)
{
 w\_dtr(base, 0xaa);
 if (r\_dtr(base) != 0xaa) return 0;

 w\_dtr(pb, 0x55);
 if (r\_dtr(pb) != 0x55) return 0;

 return 1;
}
```

[3]Pueden existir problemas si esta en modo EPP

4.3. Modo PS2

En el puerto paralelo original el puerto de datos se diseño como solo escritura. Aunque se puede usar lo bits de status para realizar un lectura, es necesario realizar dos ciclos para conseguir leer un byte.

En 1987 IBM introdujo sus modelos PS2 que entre otras características permitía que el puerto paralelo operara en modo bidireccional. Fue la primera especificación que permitía leer 8 bits en cada ciclo, no obstante no han aparecido muchos periféricos que utilicen este sistema de conexión. Desde la aparición de los modos EPP y ECP ha perdido funcionalidad pues no permite transmitir a la misma velocidad que los nuevos modos.

Con la intención de mantener compatibilidad los puertos que pueden operar en el modo PS2, por defecto se configuran en *Output mode*.

En el estándar IEEE1284 [IEE95b] se referencia a este modo como *Byte Mode*. A parte de cambiar el nombre también especifíca la señalización. Hasta la aparición del estandar existian diversas variantes, puesto que IBM no publicó una señalización concreta.

4.3.1. Detección

Teóricamente activando el bit de dirección nos permite leer valores de la linea de datos. Si en modo lectura se escribe algún valor no es posible leerlo, de esta sencilla manera se puede detectar si un puerto soporta el modo PS2.

```
int PS2\_supported(int base)
{
  int ok = 0;

  w\_ctr(base, r\_ctr(base) | 0x20); /* Dirección lectura */

  w\_dtr(base, 0x55);
  if (r\_dtr(base) != 0x55) ok++;

  w\_dtr(base, 0xaa);
  if (r\_dtr(base) != 0xaa) ok++;

  return ok;
}
```

Algunos puertos que soportan nuevos modos como EPP y ECP, permiten trabajar en PS2 mediante emulación. Para ver como activar la emulación PS2 se ha de modificar el registro **ECR** usado en el modo ECP.

4.3.2. Operación

Una vez que el puerto paralelo opera en modo PS2 para cambiar de *Output mode* a *Input mode* se ha cambiar el valor de **Direction**. Para mantener compatibilidad se han

de cambiar los bits 5 y 7 del registro **control**, pues algunos chips utilizan el bits 7 en vez del 5 para gestionar la dirección.

4.4. Modo EPP

Las siglas EPP corresponden a *Enhanced Parallel Port*, es un desarrollo de Intel, Xircom y Zenith para conseguir un puerto paralelo de altas prestaciones. Desde el principio se diseño pensando en la compatibilidad con el puerto en los modos SPP y PS2.

El modo EPP ofrece muchas ventajas adicionales sobre el puerto SPP y PS2, por este motivo ha sido rápidamente aceptado por diversos fabricantes. Unos años más tarde se creó el estándar 1284 que incorpora nuevos modos como el ECP y ofrece algunas modificaciones respecto al EPP tradicional. Este es el motivo por el que muchas BIOS implementan dos versiones de EPP, la EPP 1.7 y la EPP 1.9.

El puerto EPP permite comunicaciones bidireccionales, y proporciona transferencias de dos tipos de información. Puede cambiar de dirección con gran facilidad, por este motivo es ampliamente utilizado en sistemas que necesitan enviar información en ambos sentidos de forma constante.

Opera a la velocidad del bus ISA, un microsegundo aproximadamente (incluye handsake). La gran mejora de prestaciones con el modo PS2 se debe a que el hardware se encarga de la negociación para cada transferencia. En SPP y PS2 es necesario que el software realice el handsake, mientras que en EPP el handsake lo realiza el hardware. En SPP y PS2 son necesarios cuatro ciclos, mientras que en EPP solo es necesario un ciclo. El efecto de esta mejora se observa con implementaciones de EPP que consiguen transferencias desde 500Kbytes hasta los 2Mbytes que es el máximo teórico.

El protocolo EPP proporciona 4 tipos de transferencia de información.

- Escritura de datos, *Data Write*
- Lectura de datos, *Data Read*
- Escritura de direcciones, *Address Write*
- Lectura de direcciones, *Address Read*

Normalmente los ciclos de datos se utilizan para pasar información entre el ordenador y el periférico. Mientras que las direcciones se utilizan para pasar comandos e información de control.

Cuando el puerto paralelo opera en modo EPP posee los registros de la tabla 4.4.

Aunque no es necesario que los registros *EPP_data1-3* operen como extensión del registro de datos, esto acostumbra a ser lo más corriente. Los chips que soportan la extensión del registro permiten realizar una transferencia de 16bits o 32bits en un solo ciclo.

4.4.1. Detección

Antes de intentar detectar la existencia del modo EPP, se ha de haber detectado que se soporta el modo SPP, si no existen el modo SPP, seguro que no existe le modo EPP.

Con la intención de averiguar si el puerto paralelo soporta el modo EPP, se utiliza el bit de timeout del registro status.

Nombre	Offset	Descripción
data	0x000	Lectura o escritura en las lineas de datos sin handsaking
status	0x001	Registro Status (bit 0 indica timeout)
control	0x002	Registro de control tradicional
EPPD_addr	0x003	Lectura o escritura de la dirección usando el handsaking del modo EPP
EPP_data0	0x004	Lectura o escritura de datos usando el handsaking del modo EPP
EPP_data1	0x005	Dependiente del fabricante, puede operar como un registro con 16bits de datos o registro de control
EPP_data2	0x005	Dependiente del fabricante, puede operar como un registro con 24bits de datos o registro de control
EPP_data3	0x005	Dependiente del fabricante, puede operar como un registro con 32bits de datos o registro de control

Cuadro 4.4.: Registros en el modo EPP

Si es posible generar un timeout se activará el bit de timeout, esto es una indicación clara de que el modo EPP esta operativo. El timeout se activa despues de 10 microsegundos según las especificaciones.

La siguiente rutina retorna cierto si detecta el puerto paralelo:

```
int EPP\_supported(int base)
{
  /* Si no es posible borrar el timeout, seguro que no existe */
  if (!epp\_clear\_timeout(base))
    return 0;

  w\_ctr(base, r\_ctr(base) | 0x20);
  w\_ctr(base, r\_ctr(base) | 0x10);
  epp\_clear\_timeout(base);

  r\_epp(base);
  udelay(10);  /* Espera 10 uS de timeout */

  if (r\_str(base) & 0x01) {
    epp\_clear\_timeout(base);
    return 1; /* Detectado */
  }

  return 0;
}
```

Algunos puertos que soportan el estándar IEEE1284 [IEE95b] permiten operar con varios modos, por este motivo para detectar si existe el modo EPP es necesario activar el modo EPP mediante la modificación del registro ECR.

4.4.2. Operación

Cuando se realizan transmisiones en el modo EPP, se han de tener en cuenta el bit de timeout, así como la dirección de transmisión. Teóricamente no es necesario gestionar el bit de dirección, pero en muchos chips si que es necesario, por este motivo es conveniente tenerlo en cuenta.

En la realización de la escritura, se han de realizar los siguiente pasos:

1. Eliminar el bit de timeout si es que existe.
2. Se ha de forzar que *nInit* se mantenga cierto mientras que todos los otros bits del registro de control son falsos.
3. Escribir los valores que se desean transmitir.
4. Forzar en el registro de status que *nInit* y *SelectIn* sean ciertos, mientras que el resto es falso.
5. Verificar que no se ha producido un timeout y que el bit de *nbusy* es cierto.

El siguiente fragmento de código envía un buffer:

```
int write\_epp(int base,char *buffer, int size)
{
  int conta;

  if( r\_str(base) & 0x01)
    epp\_clear\_timeout(base);

  w\_ctr(base, 0x04); /* nInit */

  for(conta=0;conta<size;conta++)
    w\_epp(base,buffer[conta]);

  w\_ctr(base, 0x0c); /* SelectIn, nInit */

  if(r\_str(base) & 0x01)
    return -1;  /* Timeout error */

  if(!(r\_str(base) & 0x80))
    return -1   /* Busy ?? */
```

```
  return size;
}
```

La lectura es igual a la escritura con la única diferencia que al modificar el registro de control ha de forzar que el bit de dirección sea cierto [4].

Especial consideración ha de tener la eliminación del timeout en el modo EPP, existen diversas formas de eliminar el timeout. Según el fabricante existen las opciones:

- Una doble lectura del registro de status
- Escribir 0x01 en el registro de status
- Escribir 0x00 en el registro de status

Lo ideal es tener una función que elimine el timeout realizando todas las operaciones:

```
int epp\_clear\_timeout(int base)
{
  if (!(r\_str(base) & 0x01)) return 1;

  /* To clear timeout some chips require double read */
  r\_str(base); r\_str(base);

  if (!(r\_str(base) & 0x01)) return 1;

  w\_str(base, r\_str(base) | 0x01); /* Some reset by writing 1 */

  if (!(r\_str(base) & 0x01)) return 1;

  w\_str(base, r\_str(base) & 0xfe); /* Others by writing 0 */

  return !(r\_str(base) & 0x01);
}
```

[4]Forzar la dirección a cierto se logra poniendo a 1 el bit 5 del registro de control

4.5. Modo ECP

Las siglas ECP corresponden a *Extended Capabilities Port*, representa el modo de funcionamiento más moderno en el que puede operar el puerto paralelo.

Al igual que en EPP en el modo ECP el puerto paralelo opera a la velocidad del BUS AT. Normalmente el modo ECP incorpora un buffer de 16Bytes que puede usarse para leer o enviar datos [5].

Una gran diferencia con respecto al modo EPP es que el modo ECP permite operar con DMA, de forma que ya no ha de ser la CPU quien se encargue de transmitir los datos.

El estándar ECP fue creado por Microsoft y Hewlett Packard. Fue presentado a la sociedad IEEE para que creara un estándar público con toda la documentación necesaria [IEE95b]. No obstante al igual que ocurre en el EPP muchas cosas no están documentadas en el estándar y se han de recurrir a los DataSheets finales de cada chip.

El modo ECP incorpora numerosas novedades respecto a los modos anteriores:

- Soporta DMA para gestionar las transferencias.
- Numerosos estándares para especificar como funciona. IEEE1284 [IEE95b] [Cor93b], y las extensiones PnP de Microsoft [Mic96] [Mic94].
- Incorpora buffers de escritura y lectura.
- Soporta descomprensión RLE, y opcionalmente puede incorporar la compresión por hardware.
- Facilita la detección de recursos utilizados. DMA e IRQ [6]
- Puede llegar a emular todos el resto de los modos. SPP, PS2 y EPP [7]
- Añade tres registros de configuración extras.

4.5.1. FIFO

El modo ECP incorpora una FIFO para optimizar las transmisiones entre host y periférico. Gracias a este buffer no es necesario controlar el timeout al igual que en EPP [8]

Normalmente las FIFO son de 16 Bytes, no obstante este tamaño suele ser insuficiente para poder realizar compresión de hardware. Existe un chip de Cirrus que posee una FIFO de 256 Bytes, mientras que algunas FIFOS de periféricos tienen 64Bytes.

[5] El estándar no especifica el tamaño del buffer, por ejemplo un chip de GoldStar posee 256Bytes de buffer

[6] Aunque la realidad se complica ligeramente al detectar el DMA

[7] Emula una variante del EPP conocidas como EPP v1.9

[8] En EPP existe como máximo 10uS de timeout porque como máximo cada 15uS se ha de liberar el BUS para realizar un refresco de la RAM, en ECP el hardware se encarga de esperar todo lo necesario para enviar el byte, sin que exista timeout

La FIFO facilita considerablemente las operaciones de transmisión pues el software solo ha de controlar si queda especie libre en la FIFO durante la escritura, o si existe algún byte en la FIFO durante la lectura.

Se ha de recordar que para poder operar con la FIFO es conveniente controlar el bit de dirección (bit 5 del status).

4.5.2. Registros

Con la intención de controlar el modo ECP, se han introducido varios registros nuevos, pero a diferencia de el resto de los otros modos no se encuentran a continuación de la dirección base del puerto paralelo, sino con un offset de 0x400 con respecto a la base. La tabla 4.2 en la página 91 muestra los nuevos registros así como el offset que es necesario para acceder a ellos.

Existen tres registros ECR, cnfgA y cnfgB. Estos dos últimos registros muestran las principales características que soporta el chip [9], compresión, irq, dma...

4.5.2.1. Registro ECR

El registro ECR permite configurar todas las nuevas funcionalidades del modo ECP. Las siglas ECR provienen de Extended Control Register. Se encuentra 0x402 bytes desplazado con respecto con la base del puerto paralelo, por este motivo puede encontrarse en las direcciones 0x77A o 0x67A, pues con la base 0x3BC no puede existir ni el modo ECP ni el modo EPP.

7:5 Bits de modos en los que puede operar el chip.

- **000** **SPP**. Se comporta como el modo SPP, es el valor por defecto para mantener compatibilidad. Es de implementación obligatoria.
- **001** **PS2**. Emula el modo PS2, según como se configure la BIOS puede que el chip no soporte esta emulación por lo que es conveniente realizar la detección del modo PS2 una vez se ha modificado el registro ECR.
- **010** **FIFO Mode** Un modo curioso que se comporta igual que el modo SPP, pero que permite operar con el canal de DMA y usar la FIFO igual que si se tratara del modo ECP.
- **011** **ECP** El modo ECP en si mismo, algunos chips no permiten que se modifique el valor de dirección mientras que se opera en modo ECP, es necesario cambiar al modo SPP o PS2. Por este motivo cuando se desea cambiar de dirección se ha de esperar a que se vacíe la FIFO, cambiar al modo PS2, cambiar el bit de dirección y volver al modo ECP.
- **100** **EPP** Depende de como se configure la BIOS este modo no opera correctamente. Por eso al igual que ocurre con la emulación PS2, se ha de tratar de detectar el modo EPP una vez se ha modificado el valor de ECR. A diferencia

[9] La tabla 4.2 muestra información detallada de los registros cnfgA y cnfgB

del modo EPP, la emulación EPP tiene un comportamiento distinto con la señal de Busy. La versión de emulación se conoce como EPP v1.9.

101 Reservado Algunos chips los utilizan como modo especial.

110 TEST Mode En este modo los datos pueden ser escritos y leídos de la FIFO sin que exista negociación para transmitirlos en el BUS. No obstante los chips de National Semiconductors no soportan este modo.

111 Config Mode En este modo se puede acceder a los registros de configuración **cfngA** y **cnfgB**.

4 nErrIntrEn este bit solo es valido en el modo ECP. El modo ECP permite que exista otra forma de generar las interrupciones. Si es falso y se produce una transición de cierto a falso en *nFault* se genera un interrupción. Hay que tener especial consideración con estas interrupciones pues también se puede generar una interrupción si el valor de *nFault* es falso y se escribe a falso en valor de *nErrIntrEn*.

3 dmaEn

1 Permite que se empiecen las transferencias de DMA en el momento que el valor de *serviceIntr* sea falso.

0 Desactiva el canal de DMA.

2 serviceIntr

1 Desactiva el DMA y cualquier interrupción que se pueda producir.

0 Activa las transferencias de DMA, y permite que se genere una interrupción según el nivel de ocupación de la FIFO. Si el bit de dirección y el dmaEn son falsos se puede producir una interrupción cuando existen *writeIntrThreshold* bytes libres en la FIFO. De la misma manera si el bit de dirección es cierto y dmaEn es falso se produce una interrupción cuando *readIntrThreshold* bytes han sido recibidos y se encuentran en la FIFO. [10]

1 full Indica que no queda espacio en la FIFO.

0 empty No existe información el la FIFO.

4.5.3. Compresión

El modo ECP soporta compresión del tipo RLE, Run Length Encoding [11], la que puede reducir el numero de bytes necesarios para transmitir un bloque de datos. Este tipo de compresión solo es efectiva cuando existen secuencias de bytes idénticos, consiguiendo factores de compresión de 1:64.

[10] El valor de readIntrThreshold y writeIntrThreshold se han de calcular en el modo de TEST, pues depende del chip. Por ejemplo SMC proporciona un software de control que permite modificar el valor readIntrThreshold y writeIntrThreshold

[11] Para una más amplia explicación de la compresión RLE se puede recurrir a [Nel91]

En vez de transferir bytes idénticos individualmente, el dispositivo envía una byte de dirección indicando la cantidad de veces que ha de repetirse el byte que a continuación se envía.

Solo algunos chips soportan la compresión por hardware, mientras que la descompresión ha de ser soportada por todos para cumplir con el estándar IEEE1284 [IEE95b].

Es posible realizar la compresión por software y que el hardware se encargue de la descompresión, para conseguirlo se ha de escribir en el registro de datos la cantidad de bytes que se han de repetir, y despues escribir en la FIFO el byte.

Por Ejemplo:

```
w\_fifo(base,0x05); /* 1 */
w\_fifo(base,0x01); /* 2 */
w\_fifo(base,0x01); /* 3 */
w\_fifo(base,0x01); /* 4 */
w\_fifo(base,0x01); /* 5 */
w\_fifo(base,0x03); /* 6 */
```

Es equivalente a realizar:

```
w\_fifo(base,0x05); /* 1 */
w\_dtr(base,0x04);  /* 2 */
w\_fifo(base,0x01); /* 3 */
w\_fifo(base,0x03); /* 4 */
```

Con lo que se ahorran 2 ciclos.

4.5.4. DMA

La DMA permite realizar operaciones de salida y entrada de forma autónoma, sin la necesidad de usar la CPU para transferir la información. Para realizar una operación de escritura, la CPU ha de inicializar los registros del controlador de DMA con la dirección base del buffer y tamaño que posee. Cuando la transferencia finaliza se genera una interrupción para avisar al procesador que se ha finalizado la transferencia.

Algunos chips, SMC entre ellos [SMC], no permiten cualquier tamaño de DMA para realizar transferencias [12], limitan el tamaño máximo a transmitir a 32 bytes. Esto es debido a que como máximo cada 15uS se ha de realizar un refresco de la memoria, al limitar a 32 Bytes existe tiempo suficiente para realizar un refresh.

IBM con la especificación PS2 type 3 utilizó por primera vez el DMA para los puertos paralelos, pero este estándar nunca llegó a tener aceptación en el mercado, por este motivo no se ha comentado hasta el momento. La primera versión con soporte para DMA en el puerto paralelo con amplia aceptación corresponde al modo ECP.

Normalmente se ha de configurar mediante la BIOS el canal de DMA a utilizar, por este motivo es conveniente fijarse en las opciones de BIOS si se puede seleccionar el

[12] Si se conoce sobre que chip se realiza el diseño es conveniente leer el datasheet para observar si posee alguna limitación de tamaño para operar con el DMA

canal de DMA. No obstante con la nueva especificación PnP de Microsoft, se asigna automáticamente el canal de DMA sin necesidad de preocuparse si está usado por otro dispositivo.

Según el estándar IEEE1284 [Cor93b], pueden existir canales de DMA de 16bits [13] en modo ECP, canales 7,6 o 5. No obstante normalmente solo se encuentran los canales 3,2 o 1 como canales de DMA válidos [14]. Es conveniente ver la sección 4.5.5 para saber como detectar que canal de DMA se está usando.

A continuación se realiza un ejemplo de lectura usando la DMA en Linux.

```
w\_ecr(base,0xc0);  /* ECP Mode */
w\_ctr(base,0x04);  /* dirección input */

disable\_dma(dma);
clear\_dma\_ff(dma);
set\_dma\_addr(dma, virt\_to\_bus(buffer));
set\_dma\_count(dma, size);
set\_dma\_mode(dma,DMA\_MODE\_READ);

w\_ecr(base,0xc8); /* ECP Mode + dmaEn */

enable\_dma(dma);
```

Se ha de usar la misma secuencia que la del ejemplo anterior, pues en algunos chips no funciona si no se activa el dmaEn despues de programar el canal de DMA.

4.5.5. Detección

El modo ECP a diferencia del resto de los modos está integrado con la especificación PnP de Microsoft. Por este motivo facilita la detección del hardware utilizado. Cuando un chip soporta el modo ECP, es "fácil" detectar el canal de DMA, así como la IRQ usada por el puerto paralelo.

Primero se detecta la existencia del registro ECR, si este existe puede que exista el modo ECP, y también será posible intentar detectar la irq así como el canal de DMA.

Si existe el registro ECR es conveniente probar todos los modos para verificar su existencia, pues existen muchas implementaciones o configuración que solo permiten un subgrupo.

4.5.5.1. Detección del registro ECR

Existe un problema a la hora de detectar si existe el registro ECR, algunos ordenadores XT poseen alias de los dispositivos de entrada y salida en offsets de 0x400, pues solo

[13] Personalmente no conozco ningún chip en el mercado que soporte estos canales de DMA, por lo que no he podido realizar pruebas de rendimiento

[14] Normalmente desde la BIOS solo los canales 1 o 3 se pueden seleccionar

usan 10 lineas de direcciones para seleccionar el chip. Por este motivo en vez de acceder al registro de ECR puede que se esté realizando pruebas sobre el registro control.

Una vez se está seguro que no se trata del registro control, al escribir 0x34 en el registro ECR se ha de leer 0x35. El siguiente fragmento de código detecta la existencia del puerto paralelo.

```
int ECR\_present(int base)
{
  int r = r\_ctr(base);

  if ((r\_ecr(base) & 0x03) == (r & 0x03)) {
    w\_ctr(base, r ^ 0x03 ); /* Toggle bits 0-1 (autoFD y strobe)*/

    r= r\_ctr(base);
    if ((r\_ecr(base) & 0x03) == (r & 0x03))
      return 0; /* Sure that no ECR register exists */
    }

  w\_ecr(base,0x34);
  if (r\_ecr(base) != 0x35)
    return 0;

  w\_ecr(base,0x0c); /* Reset ECR */
  return 1; /* Existe el registro ECR */
}
```

4.5.5.2. Detección del modo ECP

Aunque teóricamente si existe el registro ECR tiene que existir el modo ECP, la realidad no se acerca a la teoría. Los chips LGS, más conocidos como Winbond, pueden ser configurados por la BIOS para soportar el registro ECR para emular otros modos (PS2 y EPP), y no funcionar en el modo ECP o TEST. Para detectar estos casos se conmuta al modo TEST y se escriben 4 bytes, si el bit de empty continua activo, es seguro que no soporta el modo ECP.

La siguiente función detecta si existe el modo ECP.

```
int ECP\_supported(int base)
{
  int i;

  /* If there is no ECR, we have no hope of supporting ECP. */
  if (!ECR\_supported(base))
    return 0;

  w\_ecr(base, 0xc0); /* TEST FIFO */
```

```
  for (i=0; i < 4 && (r\_ecr(base) & 0x01); i++)
    w\_fifo(base, 0xaa);

  w\_ecr(pb, 0x0c);

  return (i != 4);
}
```

4.5.5.3. Detección del canal de DMA

Detectar el canal de DMA es fácil, despues de verificar que soporta el modo ECP, se programan todos los canales de DMA, desde el 1 al 7, para que realicen una operación de lectura.

Se programa el modo TEST y se activan todas las DMAs tal y como se explica en 4.5.4, despues de pocos microsegundos [15] se para el DMA y se leen los valores de los residuos del controlador de DMA. Cuando el residuo es distinto al valor del buffer programado se ha detectado el canal de DMA.

Se ha de observar que algunas placas no implementan correctamente la DMA [16], por lo que no se ha de suponer que exista canal de DMA.

4.5.5.4. Detección de la IRQ

Detectar la IRQ es todavía más fácil que detectar el DMA. Se ha de realizar el siguiente proceso:

1. Resetear la FIFO, esto es posible conmanto al modo SPP.
2. Forzar que el bit de dirección sea falso. Se ha de recordar que este cambio es conveniente realizarlo en el modo SP2.
3. Activar el modo TEST de la FIFO y activar *nErrIntrEn*.
4. Escribir en la FIFO hasta que esté llena, antes de llenarse es seguro que se ha de producir una interrupción.

4.5.6. Operación

Las operaciones de transmisión con el modo ECP son más sencillas que con EPP pues no es necesario controlar el bit de timeout, no obstante el modo ECP permite realizar transferencias con y sin DMA.

La utilización del DMA puede traer problemas, todo depende del chip utilizado, por este motivo no es conveniente utilizarla a no ser que sea necesario o se conozca

[15] 500uS son más que suficientes

[16] La placa MP064 de Soyo no soporta DMA en el modo ECP aunque se puede seleccionar en la BIOS, posiblemente se trata de un problema de diseño

el chip a utilizar. Actualmente el problema de los chips SMC ya no existe (máximo 32Bytes de transferencia), no obstante en el mercado de PC actuales (1997), se encuentra ampliamente aceptado [17].

Las operaciones de lectura y escritura son muy sencillas, solo se ha de controlar los bits de *full* y *empty* para realizar la transferencia.

Existe un bug en los chips primeros chips SMC66x, la propia Microsoft creadora del estándar dice que se ha de desactivar los drivers de ECP en Windows 95 cuando la placa dispone de este chip. Este bug impide una correcta gestión de la transmisión y aleatoriamente pierde información. No obstante existe una solución, a la hora de enviar un byte, en vez de escribir mientras no este llena la FIFO se ha de escribir solo cuando este vacía, una vez transmitido el byte, se ha de cambiar al modo *Config* y despues volver al modo ECP, de esta forma se puede aprovechar las transferencias DMA, aunque se pierden prestaciones. Es conveniente soportar este sistema de operación y que se pueda activar si se detecta que existe el chip de SMC.

Se puede detectar la existencia del chip SMC66x mediante la siguiente rutina.

```
int SMC\_present()
{
  int val=0xff;

  if( inpb(0x3f0) == 0xff ){
    outb(0x3f0,0x55);
    outb(0x3f0,0x55);

    val=inpb(0x3f0);

    outb(0x3f0,0xaa);
  }

  return val == 0x00;
}
```

[17] No dispongo de información estadística, pero personalmente he encontrado más chips de SMC con este BUG que de cualquier otra marca

Capítulo 5 ❑

Sistemas de interconexión

Existen multitud de sistemas de interconexión entre unidades de proceso, en este capítulo solo se pretende introducir de las principales opciones que se encuentran en el mercado. Los sistemas de interconexión analizados son el puerto paralelo, USB, SCI, ATM y Myrinet.

5.1. Introducción

A la hora de seleccionar un sistema de interconexión se ha de clarificar con antelación cuál es el entorno donde funcionará.

Para poder tener un criterio de selección es necesario utilizar un conjunto de unidades que permiten evaluar las características del sistema de interconexión. En todo análisis se ha de considerar:

- El *throughput* o **ancho de banda** que soporta el sistema de interconexión. [1] Existe varias consideraciones cuando se analiza el ancho de banda:
  - Cuál es el ancho de banda que soporta el sistema.
  - Cuál es el ancho de banda que máximo que puede existir entre dos nodos.
- La **cantidad de nodos** que pueden existir en la red.
- La **tolerancia a fallos** a un malfuncionamiento de un nodo de la red. [2]
- Capacidad o no de **garantizar tiempos**, es decir si es posible realizar sistemas *realtime* usando el sistema de interconexión.
- **Latencia** existente entre diversos nodos de una misma red. [3]

[1] Es posible que un sistema de interconexión suporte un ancho de banda superior al que puede existir entre dos nodos.

[2] Algunos sistemas no permiten que un nodo deje de funcionar o ocasionan la perdida de toda la información.

[3] Muchas redes dejeneran en latencia a medida que crece el número de nodos.

5.2. Myrinet

La especifiación de Myrinet está gestionada por la empresa americana Myricom, posee las patentes aunque proporciona un ámplio soporte a desarrolladores externos. Por ejemplo ha participado en la creación de drivers para Linux.

Las redes Myrinet utilizan dos canales de comunicación para interconectar dos ordenadores, donde cada uno de los canales opera en una sola dirección. Si se desea crear una red es necesario utilizar *switchs*.

Cada canal de comunicación posee 8bits de datos, y como actualmente operan a 40MHz se conseguen transferencias de 40MBytes por canal. Al poseer dos canales consigue un ancho de banda total de 80MBytes.

En una configuración con cuatro ordenadores conectados es necesario utilizar un *switch* con ocho puertos. En esta configuración si un nodo desea conectarse con otro nodo ha de generar una paquete que pasará por el *switch*.

Las redes Myrinet permiten que varios *switchs* se interconecten en cascada. No obstante por cada *switch* que pasa cada paquete se ha de añadir información de enrutamiento [4].

Las principales características de la Myrinet son:

- Permite la creación de redes locales con múltiples nodos.
- En tiempo mínimo necesario para enviar un paquete es de 420ns.
- El formato de los paquetes es, un byte de cabecera, los datos y un byte de CRC.
- Posee un ancho de banda de 80MBytes/s.

[4]Cada vez que un paquete pasa por un *switch* el primer byte es utilizado para enrutar

5.3. SCI

El desarrollo del SCI empezó a idearse por los creadores del FutureBus al observar las limitaciones que poseia.

Al observar que la organización en forma de bus no podia escalarse, optaron por una interconexión diferente. Se creo un estándar ANSI/IEEE con las características del SCI.

El SCI reduce la latencia de comunicación de forma radical en comparación con el resto de sistemas de interconexión existentes, incluido el ATM o el FiberChannel, principalmente porque el SCI facilita la eliminación de las capas de encapsulación usadas actualmente.

En realidad la red SCI proporciona una arquitectura CC-NUMA, *Cache Coherent Non Uniform Memory Access*. Cada nodo está mapeado en un espacio de direcciones distinto. Cuando un nodo desea "comunicarse" con otro nodo ha de realizar una intrucción "load" o "store" sobre la dirección donde se encuentra el otro nodo. De esta forma si dos nodos desean comunicarse de una forma segura, el operativo solo ha proporcionar acceso de lectura y/o escritura sobre direcciones de memoria que utiliza el nodo remoto.

Cada nodo posee una pequeña cache, de tamaño variable según la placa, donde almacena en la cache local los valores de "memórias o nodos" remotos.

No obstante también proporciona una API que permite generar interrupciones en el nodo remoto, de forma que se puede realizar un sistema de paso de mensajes tradicional.

El SCI está formado por un anillo unidireccional, no obstante puede operar contra *switchs* al igual que realiza la Myrinet. Es decir puede establecerse anillos entre cada nodo y un *switch*.

Las principales características del SCI son:

- Permite la creación de redes locales con múltiples nodos.
- En tiempo mínimo necesario para enviar un paquete es de $16ns + (distanciadelcable * 0{,}7 * C)$.
- El formato de los paquetes es, cuatro bytes de cabecera, cuatro bytes de tamaño, los datos y dos bytes de CRC.
- Posee un ancho de banda de 1GByte/s.

5.4. USB

Muchos de los PC actuales conectan los periféricos utilizando los conectores que se diseñadon en los principios de 1980. Muchos de estos dispositivos poseen unas características que dificultan el diseño del operativo.

El sistema de interconexión USB, *Universal Serial Bus*, se ha creado con la intención de crear un único sistema de conectar periféricos de forma sencilla y que permita una detección automática del hardware nuevo.

Las principales características de USB son:

- Permite conectar dispositivos en casada, hasta un total de 127 dispositivos, al igual que permite la especificación IEEE1284.
- Los dispositivos se pueden conectar y desconectar en caliente.
- Tal y como su nombre indica utiliza un canal serie con codificación NZRI para enviar la información.

El ancho de banda que soporta el USB es de 12MBits/s (1.5MBytes/s). A este ancho de banda se le ha de restar las cabeceras que necesita el USB.

A causa de sus características no es un sistema de altas prestaciones. No obstante puede ser muy útil una conexión TCP/IP sobre USB, pues muchos portátiles podrian aprovechar esta conexión.

5.5. ATM

La tecnologia de transmisión ATM (*Asyncronous Transfer Mode*), fué originalmente propuesta como la estructura de conmutación para B-ISDN, pero actualmente se ha desmarcado del prósito original, pasando a formar una opción desvinculada de el.

Debido a que deriva del STM (*Syncronous Transfer Mode*), veamos brevemente cuales son sus características, para así comprender más fácilmente el porque surgió ATM.

El STM, es un systema que utiliza la multiplexación por división en el tiempo, para transmitir voz y datos a grandes distancias. Los paquetes son transmitidos síncronamente en *slots* de tiempo cada $125\mu s$. El ancho de banda, se divide gerárquicamente entre canales de medida fija. Los canales son identificados según la posición que ocupan en los *slots*. Una ventaja del STM es que utiliza conmutacion de circuitos para crear una conexión entre dos puntos, que es destruida cuando la transmisión se ha completado. Sin embargo, este sistema tiene el problema de ocupar el canal y su ancho de banda, desde el momento que se crea al conexión hasta que se destruye, malgastando así el ancho total de banda en los momento que el canal no transmita información. Por tanto este sistema no es muy recomendable para las aplicaciones que realizen las transmisiones en grandes bloques de manera poco continuada.

ATM fué propuesta independientemente por Bellcore y otras compañias de telecomunicaciones europeas para poder solucionar este tipo de problemas.

El sistema de ATM, esta basada en la multiplexación por división asíncrona en el tiempo, donde su base de la teoria consiste en etiquetar los paquetes según a la conexión a la que pertenecen, y no segun el *slot* de tiempo. Utiliza tamaños de paquetes muy reducidos para altas velocidades de transmisión y muy bajas latencias, de esta manera un usuario no monopoliza la conexión, ya que circulan gran cantidad de pequeños paquetes de usuarios mezclados. El *routing* de los paquetes en los *slots* denominados *cells*, es determinado en tiempo de establecimiento, donde se crea un canal virtual entre los dos extremos. No divide el ancho de banda entre diferentes canales, pero en cambio lo da en proporción de las peticiones del usuario.

ATM és asincrono debido a que los paquetes llegan en intervalos irregulares, ya que no existe un tiempo fijo en el cual una máquina tiene que transmitirlos. De esta manera los paquetes son transmitidos cuando hay *cells* libres. En caso de que no hayan *cells* libres para transmitir un paquete, este se dejará en una cola de algun tipo, hasta que esto ocurra.

El sistema de ATM, define tres niveles para su completo funcionamiento:

- El nivel más bajo es el nivel físico. Consiste en el transporte físico utilizado para eltransporte de las *cells* de un nodo a otro. La definición de ATM es muy flexible en este tema, en cuanto a que puede trabajar con varios tipos de transporte.

- El nivel intermedio es el nivel de ATM. Este proporciona la conmutación y *routing* de los paquetes, segun sus identificadores de las etiquetas (VCI y VPI). Tambien produce la información de las cabeceras de las *cells* y la extrae de los paquetes recibidos.

- El nivel superior, es el de adaptación. Su función es controlar el *mapping* entre los diferentes tipos de tráfico y las *cells* de entrada y de salida.

Parte III.

Implementación del proyecto

Capítulo 6 ❑

Objetivos y estructuración

Una vez se posee una visión global del proyecto antes de empezar el diseño es conveniente marcarse unos objetivos claros, así como una filosofía de trabajo.

6.1. Introducción

A estas alturas del documento, se ha dado una vision global de los puntos básicos que se deben tratar para poder implementar y entender el proyecto que nos ocupa.

Por un lado, se ha visto como se estructura y funciona la *suite* de protocolos del TCP/IP, desde el el nivel más bajo hasta el nivel de aplicación. Como más adelante se verá, el proyecto de interconexión no presupone en ningún momento que solo deba funcionar con esta *suite*, es mas, el diseño es suficientemente abierto para que pueda funcionar, accediendo directamente al nivel más bajo, utilizando cualquier aplicación que utilice el TCP/IP o incluso en combinación simultánea con otros protocolos como el IPX.

Por otro lado, también se ha dado una visión básica de la interfaz física que se quiere utilizar para la interconexión como es el SCSI y el puerto paralelo. Sin pretender hacer un análisis exhaustivo de su comportamiento ni definición, se han sentado las bases y filosofía de su diseño, con las que podemos contar en la implementación.

Aunque el diseño se encuentra enfocado a utilizar el puerto paralelo y placas SCSI como sistema de interconexión, también es importante considerar como se podrían integrar otros sistemas de conexión como el SCI, la Myrinet, ATM ....

El capítulo que ha servido de conjunción de los temas anteriormente expuestos, ha sido el titulado "El sistema operativo Linux". En este capítulo, a parte de dar un breve repaso a que es Linux (el sistema operativo escogido para la implementación) y su estructura global, se ha hecho incapié en los apartados de red y de SCSI debido a que son los que deberán dominar en la implementación del proyecto.

Visto todo ello, y con los conocimientos en las áreas adecuadas, esta parte del documento proporcionará los objetivos que se quieren lograr con el proyecto, la filosofía y alternativas del diseño propuesto, y detalles de la implementación realizada. Todo ello, con sus pruebas de rendimiento y discusiones comparativas con otros métodos.

6.2. Objetivos

Los objetivos básicos que se pretenden alcanzar con este proyecto, los podríamos resumir en la siguiente frase: *"Conseguir una plataforma de interconexión entre computadoras, que tenga un alto rendimiento y un bajo coste"*. En un principio esta interconexión, se quería que fuese orientada a poder formar máquinas multicomputadores. Esta idea se quedó un poco de lado debido a que no se dispuso de presupuesto para hacerlo, aunque ya se verá que el resultado final, facilita enormemente la implementación de esta faceta.

A partir de esta premisa, y teniendo en cuenta las opciones tanto de plataformas a utilizar como de sistemas operativos sobre los cuales implementarlo, se han ido ampliando y matizando los objetivos tal y como se ira observando.

En primer lugar, una de las primeras decisiones a tomar en el inicio del proyecto, fue el tipo de plataforma de comunicación a utilizar. La primera pregunta que surgió, fue si se debía utilizar un tipo de plataforma ya existente en el mercado, o se debía diseñar y implementar un hardware específico. Debido a que la intención final del proyecto es que fuera una implementación real y utilizable por el máximo de usuarios posible, la idea de una implementación hardware quedo un poco de lado. Eso es debido a que es, evidentemente, es mucho más difícil introducir un nuevo tipo de interconexión entre máquinas, si el cliente o usuario necesita adquirir (o construir) un nuevo hardware, con el consiguiente gasto y molestias iniciales. Por tanto, se decidió que la plataforma a utilizar, debía ser una que estuviera ya introducida en mas o menos ámbito dentro del parque de máquinas de gama media-baja.

Se barajaron varias hipótesis, y se analizaron diferentes alternativas:

SCI,Myrinet Las cuales son de altas prestaciones pero tienen un elevado coste, y no están introducidas ni mucho menos, en gran medida en el mercado.

ATM A pesar de sus buenas prestaciones, el precio, su poca implantación en el momento de la decisión y la falta de recursos hicieron desestimar la opción.

SCSI Gozaba de una extensa implantación en el mercado, sobretodo en la parte de máquinas de servicios. Podía llegar a dar muy buenas prestaciones y su coste no se consideró demasiado elevado.

Ethernet 10,100 Mbits Cumplían bastante bien las especificaciones, pero las prestaciones parecían un poco justas. La especificación de la tecnología de 100 Mbits al inicio del proyecto aún no tenia implementación hecha.

Puerto Paralelo Era un candidato con un precio excepcional y con una inmejorable implantación de mercado. Su punto más débil era el rendimiento, pero las especificaciones le ponían en un nivel aceptable.

Puerto serie Su bajo rendimiento, hizo que no fuera un serio aspirante para formar una máquina de altas prestaciones, a pesar de su bajo precio y alta implantación.

De todas ellas, las alternativas que se vieron viables, tanto por cumplir los requisitos como por contar con los recursos, fueron las opciones de tarjetas SCSI y tarjetas con puertos paralelos.

Con todo, se empezó a vislumbrar que seria bueno, que el diseño fuera lo bastante flexible como para utilizar cualquiera de las dos, sin demasiados cambios, o incluso que fuera adecuado para poder adaptar tecnologías de futuro, o tecnologías actuales con futuros mejores costes.

Así pues, este fue el segundo paso, se debía diseñar un sistema suficientemente abierto o modular para soportar diferentes tipos de plataformas de interconexión con el mínimo esfuerzo.

La elección para llegar a tal meta, pasaba por escoger un sistema operativo que nos diera esas posibilidades. La decisión fue mas sencilla que la plataforma de transmisión. Evidentemente, por razones de presupuesto y recursos, se descartaron aquellos sistemas operativos de los cuales no se podía disponer de los fuentes de manera gratuita, entre los cuales se encontraban:

- Solaris y Sun/OS (Sun Microsystems)
- Windows 95, NT (Microsot)
- OSF/1 (Mach) (Open Software Foundation)
- BSD 4.x (Berkeley Software Distribution)
- HP/UX (Hewlett Packard)

Por tanto, la lista de candidatos se había reducido a dos: FreeBSD y Linux. Estos dos sistemas operativos tienen muchas características comunes, incluso hay partes de código compartidas o modificadas entre ellos. Al final se escogió Linux como el más adecuado para la implementación del proyecto. Básicamente la decisión de escogerlo frente al FreeBSD fue el hecho que se tenia un mayor conocimiento y experiencia por un lado, y por otro también se tenia más contacto con gente que implementaba y estaba relacionada con el sistema operativo Linux. A parte de estas dos razones, un factor fundamental fue la posibilidad que brindaba el Linux para trabajar con modules dentro del kernel (ver 3.3).

Vistas y analizadas las posibilidades y recursos disponibles, la selección de hardware y software, acabó siendo la siguiente:

Un conjunto de cuatro máquinas, formado por un Pentium, dos Pentiums duales y un 386. Se dispone de dos tarjetas SCSI modelo Adaptech 1542B, corriendo el sistema operativo Linux[1].

En este momento, la fase de viabilidad se había superado, se escogió una plataforma adecuada para los objetivos que se buscaban, y se consiguieron los recursos para poder realizarlo.

[1]La versión del sistema operativo fue cambiando y adaptándose a las nuevas versiones que salían al mercado. Se empezó con la versión de kernel 1.3.94.

6.3. Filosofía de diseño

Como se ha comentado en el apartado anterior, el diseño debe ser lo suficientemente flexible y modular para poderse adaptar rápidamente a nuevas tecnologías y plataformas hardware que puedan salir en el futuro, sin tener que modificar el esquema estructural. A la vez, la estructura no debe ser limitadora de factores que nuevas tecnologías puedan incorporar.

Sin perder de vista estos dos factores, el diseño estructural del sistema es modular por capas y define solo las estructuras necesarias básicas de inter-operatibilidad entre ellas, evitando así una rigidez excesiva o una limitación de base.

El sistema (dentro de kernel) se compone de tres niveles excluyentes y ordenados, similar a la filosofía que se veía en los niveles del TCP/IP.

Así pues los tres niveles definidos son los siguientes:

TDS *Threaded Support Device* es el módulo que intercepta las funciones genéricas de red, para dar una interfaz homogénea a los módulos del nivel inferior. Además, puede implementar diferentes modalidades o filosofías de procesos (*threads*, *bottom-halves*...) así como abstraer o organizar la gestión de colas de paquetes recibidos y enviados.

Type Adapter Este nivel es el que debe encargarse de gestionar la parte necesaria según el **TIPO** de plataforma haya por debajo. Debe implementar una interfaz común para TODAS las placas específicas del **TIPO** de hardware que represente, se puede decir que es un adaptador de la *familia hardware*.

Hardware Adapter Es el nivel más bajo, y es el que se debe encargar de abstraer el hardware subyacente, dando las estructuras y interaccionando con las funciones del nivel superior.

En el caso de la implementación con tarjetas SCSI, el *Type Adapter* es un módulo que define una serie de estructuras y funciones basadas en el protocolo SCSI, las cuales deben ser utilizadas por los diferentes drivers de distintas controladoras SCSI (nivel de *Hardware Adapter*). Así por ejemplo, si el nivel medio define la función y parámetros a la que se debe llamar cuando la tarjeta SCSI ha recibido un paquete, el nivel *Hardware Adapter* se deberá encargar de realizar los pasos necesarios de manejo del bus o de colas internas de comandos SCSI... pero deberá llamar a la función definida con los parámetros adecuados cuando haya recibido un paquete.

El ejemplo estructural aplicado al caso de controladoras SCSI i puertos paralelos se muestra en la figura 6.2.

Tal y como se puede observar, utilizando este tipo de estructura se consiguen las siguientes facilidades.

1. Facilidad de modificación por su diseño modular por capas

2. Compartición de código entre módulos

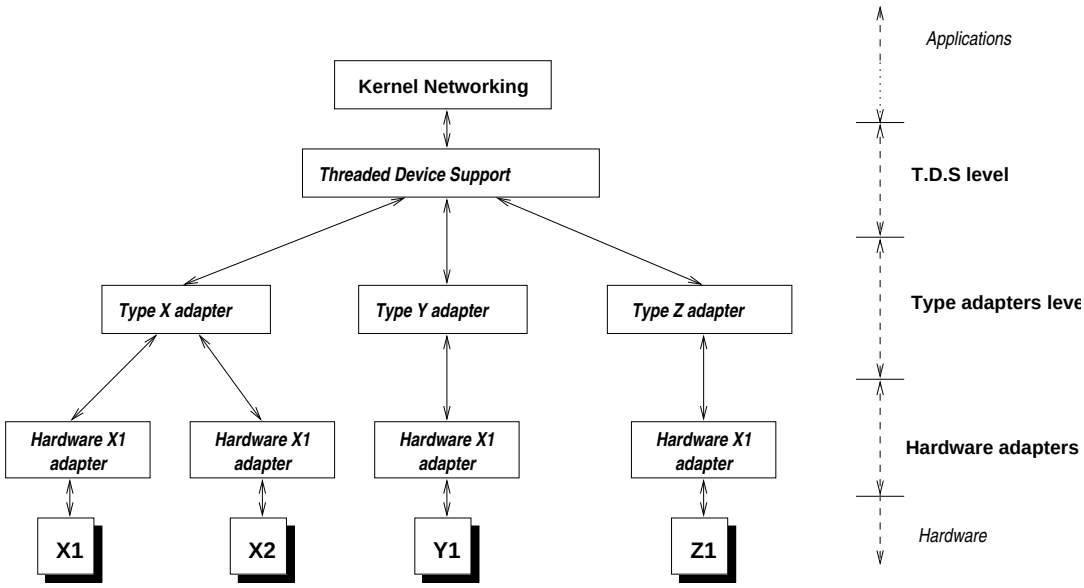

Figura 6.1.: Esquema estructural de niveles

3. Flexibilidad para la introducción de una nueva familia o tipo de hardware de interconexión.

4. Flexibilidad para la introducción de un nuevo módulo para un nuevo modelo de hardware de interconexión de una familia ya soportada.

5. Posibilidad de distintas implementaciones de gestión de paquetes centralizadas en un solo módulo, y transparente a los demás.

6. Gestión de estadísticas centralizada.

7. Posibilidad de acceder directamente a módulos inferiores desde espacio de usuario (por medio de un dispositivo simple de *filesystem*).

8. Estructura de fácil manejo y chequeo de módulos independientes, ya que todos pueden ser un *module*.

Vistas todas las características, parece claro que el diseño es adecuado y cumple con los requisitos que se habían planteado a priori.

A pesar que pudiera parecer una estructura lógica, el diseño ha soportado gran número de modificaciones y rediseño a lo largo del estudio y implementación del proyecto. Por la misma razón, se cree que incluso se puede adaptar más en el futuro. Por ejemplo, una de las fases de estudio actual incluye la reorganización del modulo del TDS para simplificar-lo aún más, y otra consiste en la asimilación o anexión de ciertas partes entre los módulos genéricos (*Type Adapters*) y drivers específicos, para aumentar la eficiencia disminuyendo las secuencias de llamadas.

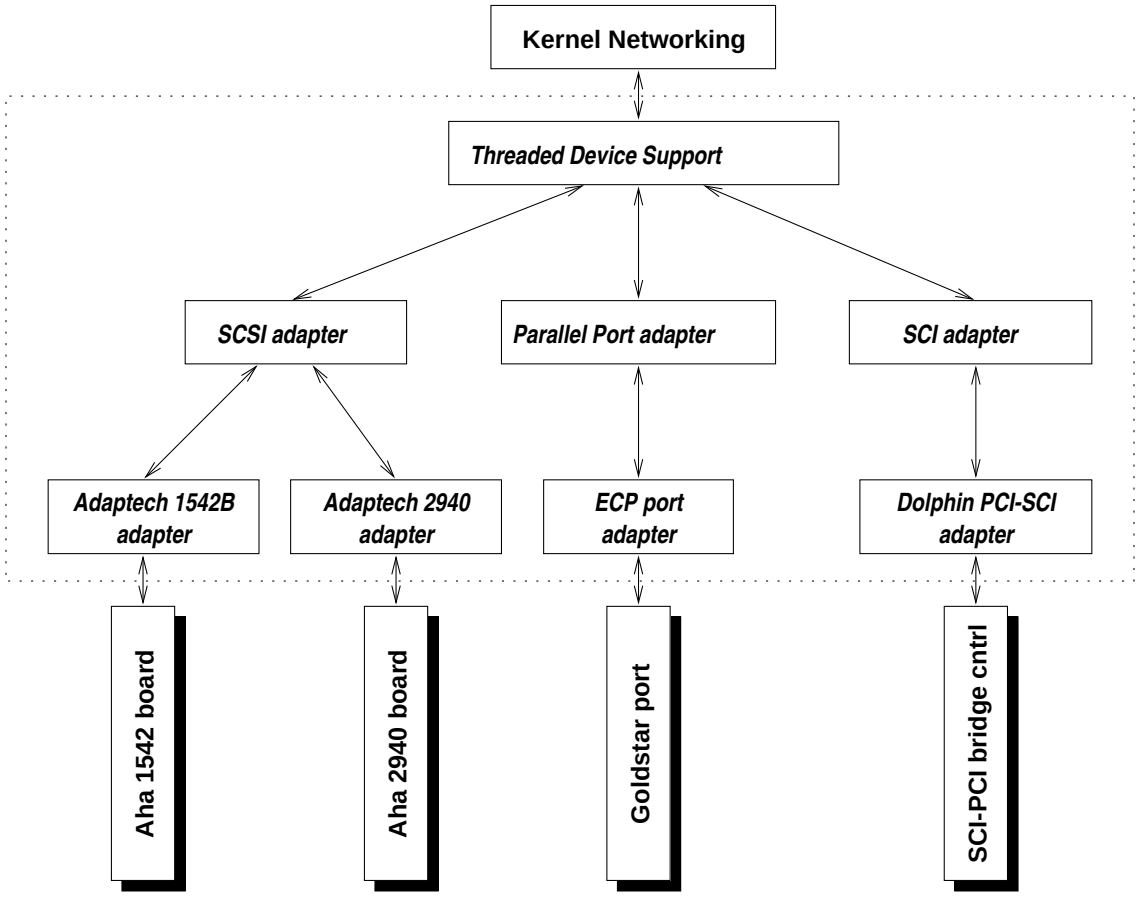

Figura 6.2.: Ejemplo de la estructura en una configuración de máquina con tarjetas SCSI, puerto paralelo y SCI

Capítulo 7 ❑

parport

En este capítulo se describen las características, así como el funcionamiento del módulo `parport`. Este nuevo módulo proporciona una API para ser usada por cualquier driver del Kernel que necesite el puerto paralelo, de igual manera incorpora la facilidad de compartir el puerto paralelo con dispositivos conectados en daisy chain.

7.1. Problemática encontrada

En las especificaciones del proyecto se indica la necesidad de implementar sobre Linux una interconexión de dos ordenadores usando los modos avanzados del puerto paralelo, es recomendable que antes de leer este capítulo se lea el capítulo que describe el puerto paralelo en general de la página 88.

Una de las normas implícitas al diseñar algún módulo del Kernel es tratar de usar todo el código ya existente, sin duplicar funcionalidades. Si se pretende que el diseño sea aceptado por la comunidad de Linux es necesario analizar cual es el estado actual, y implementar un diseño coherente con el resto de los subsistemas existentes.

Antes de empezar el diseño del puerto paralelo, se observó como gestionaban el resto de dispositivos del Kernel, la utilización del puerto paralelo. El kernel de Linux poseía las siguientes características:

- Todos los dispositivos *utilizan sus propias funciones de detección y transferencia*.
- *No permiten compartir el puerto paralelo*, en realidad si se carga un module aunque no use el puerto paralelo ningún otro dispositivo puede usar ese puerto [1].
- *Existen múltiples dispositivos que usan el puerto paralelo* como sistema de enlace:
  - **PPA** *Parallel Port Adapter* utiliza el puerto paralelo para conectar unidades *ZIP* de Iomega. Normalmente opera en *Nibble Mode*, no obstante reconoce "algunos" chips que operan con *EPP Mode*.

[1] Este problema se observa claramente cuando algún sistema se compila sin modules al cargarse desde el principio y no poder descargarse

- **PLIP** *Parallel Line IP*, permite interconectar dos puertos paralelos usando el *Nibble Mode*. Solo puede operar en nibble mode, por lo que no consigue buenas prestaciones.
- **LP** dispositivo de impresora, opera con el modo SPP.
- **ATP** *Attached eThernet Pocket*, se trata de un adaptador de red ethernet que utiliza el puerto paralelo como sistema de conexión.
- **de620** *D-link Ethernet*, adaptador de red ethernet usando el puerto paralelo como sistema de interconexión. Similar al modulo **ATP**.
- **BAYCOM** Driver para la gestión de modems *Baycom*, estos modems permiten a radio aficionados interconectarse. Se conectan al puerto paralelo.
- **BPCD** *Back-Pack CDROM* usa el puerto paralelo en modo SPP para acceder a un CDROM. No implementa el modo EPP aunque el CDROM si que soporta el modo [2]
- **EZ** Magneto óptico de SyQuest que se conecta al puerto paralelo, el dispositivo hardware puede negociar en modo EPP, aunque el driver solo soporta el modo SPP.

- Casi todos los dispositivos hardware soportan modos más rápidos que el SPP, aunque *solo el driver PPA implementa parcialmente un modo más avanzado.*

7.2. Características deseadas

Aunque originalmente no era la intención del proyecto se observó la necesidad de realizar modificaciones en el sistema de gestionar el puerto paralelo dentro del Kernel de Linux, de esta forma el proyecto tomó nuevas dimensiones. Se optó por diseñar un nuevo módulo para gestionar el puerto paralelo de forma centralizada con las siguientes características:

- *Gestor del propietario.* Un módulo central donde se cede el control del puerto parelelo al dispositivo que lo necesite.
- Una librería o module donde se centralizan las funciones de entrada y salida.
- Un sistema centralizado de detección. Cada subsistema que necesite el puerto paralelo no ha de realizar la detección por su cuenta.
- Utilizar los nuevos modos, permitiendo transferencias en modo EPP y ECP.
- Soportar las capacidades PnP con las que operan algunos chips existentes en el mercado.

[2] En realidad más que un CDROM se trata de una controladora IDE que se conecta al puerto paralelo, por lo que con pocas modificaciones es posible conectar un disco duro

- Soportar el estándar en desarrollo IEEE1284.3 [3] [IEE95b].
- Permitir una migración paulatina del sistema anterior al nuevo sistema.

De esta forma el proyecto original que consistía en implementar una interconexión entre dos ordenadores Linux queda en un segundo plano. Al ver que como una extensión lógica, el proporcionar un subsistema de puerto paralelo, el proyecto toma otras consideraciones y es necesario llegar a acuerdos con todos los desarrolladores de los módulos implicados.

Las condiciones anteriores son las usadas para realizar el diseño del puerto paralelo, no obstante cuando el diseño ya se había realizado y la implementación estaba prácticamente finalizada apareció un problema. *Un grupo de Ingleses estaba desarrollando un proyecto muy similar* al parport, ninguno de los dos grupos de trabajo había finalizado, por lo que se optó por fusionar los proyectos. Se utilizó una mailing list de puerto paralelo para discutir como se fusionarían las APIs. La dirección de la mailing list es *linux-parport@@torque.net*.

Se ha de agradecer la buena voluntad de los miembros del grupo de trabajo, así como la alta interactividad que se ha producido entre los miembros. Han existido múltiples colaboraciones, pero las principales miembros del grupo han sido:

- David Campbell, *campbell@@tirian.che.curtin.edu.au* (Australiano)
- Tim Waugh, *tmw20@@cam.ac.uk* (Alemán, estudiando Computer Science en Jesus College - Cambridge - UK)
- Philip Blundell, *Philip.Blundell@@pobox.com* (Inglés, estudiante de Computer Science en el Trinity College - Cambridge - UK)

Como consecuencia de la fusión de esfuerzos algunos opciones que habían sido planteadas han sido dejadas de lado, y otros problemas que no se habían planteado han podido ser solucionados.

El proceso de acercamiento ha sido lento y actualmente todavía continua expandiéndose el modulo parport. De las condiciones de diseño que poseía el modulo, por el momento (Mayo de 1997) no se podido acordar una API unificada de funciones de entrada y salida. [4]

[3] Permite conectar simultaneamente varios dispositivos en el mismo puerto paralelo, y existe un protocolo para seleccionar los dispositivos conectados en cadena

[4] No obstante actualmente se está volviendo a considerar la opción de incluir una función de entrada y salida para cada modo soportado por el puerto paralelo (ECP, EPP y SPP)

7.3. Diseño

Una característica de diseñar algo sobre Linux es que nunca se acaba, siempre se tiene que hacer referencia con que versión, pues no se trata de un diseño estático, sino de un diseño evolutivo. El diseño del parport proporciona una API coherente y a partir de esta API se pueden crear propuestas de ampliación o modificación.

Una vez han sido aclaradas las condiciones de diseño se han de empezar a definir las características de la implementación para que aparece en las últimas versiones de Linux (*Linux 2.1.40*). Puesto que en estos momentos las funciones de entrada y salida las ha de proporcionar cada uno de los módulos que usen el puerto paralelo como sistema de interconexión, se está considerando la posibilidad de ampliar la API con la incorporación de funciones de entrada y salida,

En el diseño del parport se puede distinguir.

- El *diseño de las estructuras necesarias* para gestionar los puertos paralelos así como los dispositivos conectados.
- Se ha de incorporar la *especificación de una API* consitente con los módulos actuales y que permita una migración paulatina.
- El conjunto de *funcionalidades que proporciona al sistema mediante el Proc Filesystem.*
- *Parámetros del módulo parport*, así como la interacción con los módulos existentes.

7.3.1. Estructuras definidas

Antes de empezar con la especificación de la API es conveniente realizar una introducción a las dos estructuras principales que se han creado.

Con la intención de gestionar el puerto paralelo existen dos estructuras, por un lado una estructura que mantiene información del puerto paralelo físico `struct parport`, mientras que otra estructura mantiene información de cada dispositivo que está conectado a dicho puerto paralelo (`struct ppd`). La figura 7.1 muestra un ejemplo donde existen dos puertos paralelos y tres dispositivos, los dispositivos podrian ser una impresora en un puerto, una unidad ZIP en el otro puerto y una conexión PLIP en daisy chain con la unidad ZIP.

struct parport Posee toda la información referente al puerto paralelo, existe una de estas estructuras por cada puerto paralelo que se detecte. Por ejemplo si tenemos dos puertos paralelos en el mismo ordenador en 0x378 y 0x278 existirán dos de estas estructuras que estarán en una lista.

Los descripción y nombre de los campos de la estructura parport se especifican en la siguientes lista:

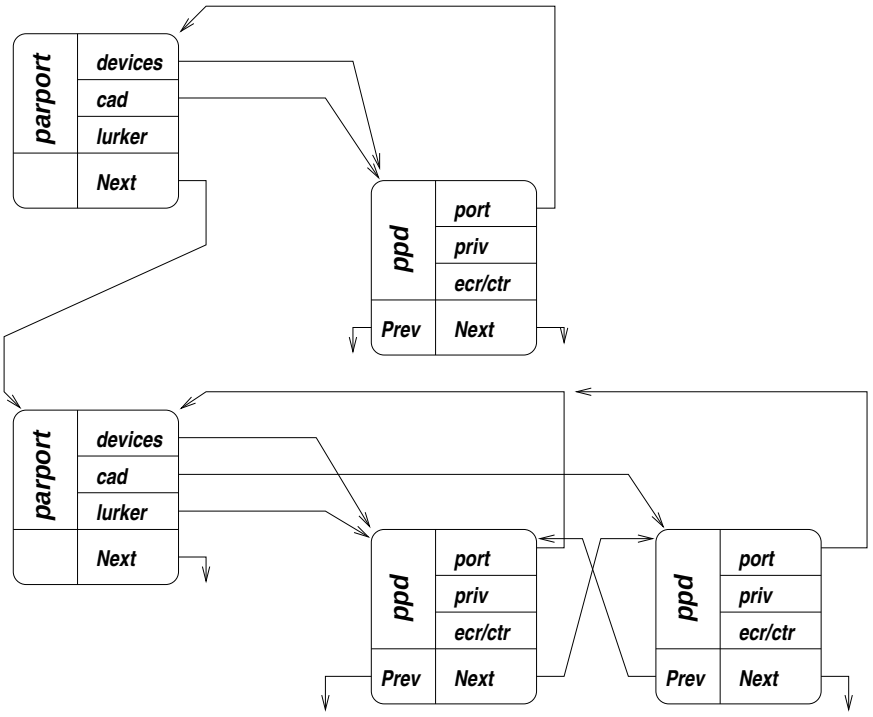

Figura 7.1.: Relación entre parport y ppd

base Dirección base donde se encuentra el puerto paralelo, en los PCs puede ser 0x3BC, 0x378, 0x278. Aunque como la gestión del puerto paralelo se pretende que funcione en máquinas Sparc y PPC no limita la dirección de ninguna manera.

size Tamaño en bytes que ocupa el puerto paralelo desde la base. Normalmente es 3, pero los puertos paralelos que soportan el modo EPP tienen un tamaño de 8Bytes.

name Nombre del puerto paralelo con el que se registran los recursos, los nombres siguen la sintaxis "parport %d", donde %d corresponde a la posición de detección. El orden de detección es 0x378,0x278 y por último 0x3BC.

irq La interrupción en la que reside el puerto paralelo, es -1 si no usa interrupciones. Esto puede ser posible por que sea imposible detectarla o que se desactive voluntariamente.

dma El canal de DMA con el que puede operar el modo ECP del puerto paralelo, es -1 si no usa DMA. Es posible por que sea imposible detectarla o no soporte el modo ECP o FIFO.

modes Modos con los que puede operar el puerto paralelo, activa un bit por cada modo que soporta. Existen unas constantes que indican que bit se activa:

PARPORT_MODE_SPP Soporta el modo SPP, actualmente todos los puertos paralelos han de soportar este modo.

PARPORT_MODE_PS2 Soporta el modo PS2.

PARPORT_MODE_ECPPS2 Puede emular el modo PS2 mediante el registro ECR.

PARPORT_MODE_EPP Soporta el modo EPP v1.7

PARPORT_MODE_ECPEPP Soporta el modo EPP v1.9

PARPORT_MODE_ECP Soporta el modo ECP

PARPORT_MODE_ECR Tiene el registro ECR.

devices Puntero a la lista de dispositivos que se han registrado en el puerto paralelo. La lista de dispositivos se explica en la estructura `ppd`.

cad Puntero al dispositivo que actualmente está operando con el puerto paralelo, por ejemplo una impresora mientras que está imprimiendo.

lurker Con la intención de poder integrar antiguos dispositivos de puerto paralelo como el PLIP, se ha creado el tipo lurker, los dispositivos que se registran como lurker. Como su nombre indica son dispositivos que "están al acecho", es decir son los últimos que se avisan y gestionan todas las señales que no procesan el resto de dispositivos. Esto ocurre pues el módulo PLIP posee una señalización no compatible con el estándar IEEE1284 y puede recibir interrupciones del ordenador remoto sin previo aviso. Solo puede existir un dispositivo lurker por cada parport.

next Apunta a la siguiente estructura parport que se ha detectado, si no existen más puertos paralelos apunta a NULL.

flags Flag usado para evitar que pueda liberarse dos veces el mismo recurso.

pdir Estructura necesaria para la gestión del Proc Filesystem.

probe_info Información que se ha podido extraer de los dispositivos que se encuentran conectados al puerto paralelo. Esta información se extrae mediante la negociación definida en el estándar IEEE1284.

speed Resultado de un mini benchmark que muestra la velocidad máxima a la que puede operar el puerto paralelo. Este benchmark mide la velocidad de lectura de la FIFO del puerto paralelo a memoria usando el DMA en modo TEST.

struct ppd El nombre de la estructura `ppd` proviene de *Parallel Port Device*, tal y como su nombre indica gestiona los dispositivos que se conectan a cada parport, puerto paralelo.

Cuando el kernel quiere registrar un nuevo dispositivo, impresora, ZIP ..., crea una nueva estructura `ppd` que asocia a un puerto paralelo. La estructura `ppd` posee la siguiente información:

name Nombre del dispositivo, este nombre se proporciona por el dispositivo a la hora de registrarse.

port Puntero a la estructura parport a la que se encuentra asociada el dispositivo.

preempt Función que ha de proporcionar el gestor del dispositivo, se llama cada vez que el gestor del puerto paralelo quiere que el dispositivo ceda el control del puerto. El dispositivo tiene que retornar cero si puede ceder el control del puerto. Si posee alguna operación en curso y no puede ceder el control retorna distinto de cero.

wakeup Función llamada por el gestor del puerto paralelo para avisar al dispositivo que puede adquirir el control del puerto. Si el dispositivo está interesado en el control del puerto retorna cero, si no necesita el puerto retorna distinto de cero. Se ha de recordar que si el dispositivo es de tipo LURK es el último en ser avisado pues es casi seguro que quiere adquirir el control del puerto paralelo.

private Parámetro que pasa a las funciones `preempt` y `wakeup`, se puede utilizar para facilitar la gestión de múltiples dispositivos.

flags Características del dispositivo que está conectado al puerto paralelo, actualmente solo existe el flag `PARPORT\_DEV\_LURK`, este flag indica que se trata de un dispositivo que se encuentra en escucha, es característico de los módulos PLIP y EPLIP. Un puerto paralelo solo soporta un único dispositivo lurker. Cuando un dispositivo es lurk está apuntado por el puntero lurker de la estructura parport.

ctr/ecr Valor del registro de control y ECR, se utiliza para almacenar el valor del los registros cada vez que se cambia el control entre distintos dispositivos.

next/prev Punteros que permiten mantener una lista bidireccional de dispositivos que pertenecen al puerto paralelo.

7.3.2. Especificación de la API

La API sobre la que se ha conseguido un acuerdo posee un total de nueve funciones. Aunque actualmente se está pensando en expandirse para añadir funciones de entrada y salida así como soportar todas las fases de la negociación del estándar IEEE1284.

struct parport *parport_enumerate(void); Retorna un puntero al inicio de la lista de todos los puertos paralelos que se han detectado en la máquina.

struct parport *parport_register_port(unsigned long base, int irq, int dma); Registra un nuevo puerto paralelo en la dirección proporcionada. Pueden existir tres opciones:

- Si en la dirección base proporcionada ya se ha detectado un puerto paralelo retorna la estructura ya existente.

- Si detecta un nuevo puerto lo añade a la lista del sistema y retorna un puntero a la estructura parport creada.
- Si es incapaz de encontrar un puerto paralelo en la dirección indicada retorna NULL.

No es muy normal llamar a esta función, lo más coherente es llamar a `parport\_enumerate` que retorna una lista de puertos detectados. No obstante esta función facilita la migración de antiguos dispositivos.

int parport_in_use(struct parport *port); Retorna cierto si el puerto paralelo está siendo usado por algún dispositivo.

void parport_destroy(struct parport *port); Elimina todos los recursos utilizados por el puerto paralelo, irq, dma y region. Retorna falso si existe algún dispositivo registrado en el puerto paralelo.

Esta función suele utilizarse para que los dispositivos antiguos que utilizan el puerto paralelo puedan liberar el puerto paralelo en uso y poder registrarse. De esta forma se permite una convivencia total con el sistema anterior de gestión de puertos paralelos.

struct ppd *parport_register_device(struct parport *port, const char *name, callback_func pf, callback_func kf, irq_handler_func irq_func, int flags, void *handle); Registra un nuevo dispositivo dentro del puerto paralelo pasado como parámetro en **port**. Proporciona toda la información que necesita el Kernel para operar con el puerto paralelo. Retorna un puntero a la estructura `ppd` creada para gestionar el device.

Los parámetros que pasa son:

port Puntero al puerto paralelo sobre el que se registra el nuevo device.

name Puntero al string que posee le nombre del nuevo dispositivo, el device ha de reservar la memoria necesaria para almacenar este nombre.

pf Función de `preempt` que proporciona el dispositivo para gestionar el device [5]. Esta función es llamada para que el puerto ceda el control del puerto paralelo.

kf Función de `wakeup` que proporciona el dispositivo, esta función se llama cada vez que el gestor del puerto paralelo considera que el dispositivo puede tomar el control del puerto.

flags Características de comportamiento que ha de poseer el dispositivo registrado, actualmente solo existe **PARPORT_DEV_LURK**.

handle Parámetro que pasará cada vez que se llamen las funciones `preempt` y `wakeup`.

[5] callback_func es un typedef del tipo `typedef int (*callback_func) (void *);`

void parport_unregister_device(struct ppd *dev); Es la función inversa a parport_register_device, elimina la estructura correspondiente al dispositivo conectado al puerto paralelo. [6]

int parport_claim(struct ppd *dev); Intenta de ceder el control del puerto paralelo al dispositivo que la está llamando. Si el puerto paralelo está siendo utilizado por otro dispositivo llama a preempt para que lo abandone. Si no consigue el control del puerto para el nuevo dispositivo retorna distinto de cero. Retorna cero si consigue el control para el nuevo gestor del puerto paralelo.

void parport_release(struct ppd *dev); Cuando un dispositivo ha finalizado la temporalmente la utilización del puerto paralelo llama a esta función, que intentará ceder el puerto paralelo a algún dispositivo que lo necesite dentro de la lista de existentes. Una vez se ha realizado un parport_release si se necesita el puerto paralelo es necesario realizar un parport_claim.

int parport_ieee1284_nibble_mode_ok(struct parport *port, unsigned char mode); Realiza la fase de negociación especificada en el estándar IEEE1284, poniendo en el bus el valor de mode. Si la negociación se ha realizado exitosamente retorna distinto de cero.

Esta función suele utilizarse para detectar la existencia de dispositivos que soportan el estándar IEEE1283 [Cor93b].

7.3.3. Module

El nuevo sistema de parport ha de permitir operar como un module, ver 3.3 en la página 3.3 para más información sobre los modules de Linux. También ha de soportar que como parámetros se puedan configurar los puertos existentes.

El modulo parport ha de soportar los siguientes parámetros:

io Admitir una lista de direcciones base separadas por comas sobre las que intentará detectar puertos paralelos. De esta forma si existe algún hardware especial que mapea puertos paralelos en direcciones no estándar pueden ser añadidos al sistema con facilidad [7].

irq Lista de interrupciones que asigna a los puertos pasados como parámetro en `io`.

dma Lista de canales de DMA que asigna a los puertos pasados como parámetro en `io`.

Por ejemplo la siguiente linea de código indica que existen dos puertos paralelos en 0x378 y 0x278, que 0x378 usa la interrupción 7 y 0x278 usa la interrupción 5.

```
#insmod parport.o io=0x378,0x278 irq=7,5
```

[6] Se ha de observar que la memoria se ha declarado del tipo GFP_KERNEL por lo que no será posible ni registrar ni desregistrar el dispositivo dentro de una interrupción

[7] El chip de HOLTEK [Hol] permite mapear el puerto paralelo en direcciones no estándar

También es posible indicar los parámetros en el LILO, para más información sobre el lilo se puede consultar [Almb] y [Alma]. Operando desde el lilo los parámetros se modifican ligeramente, por ejemplo si se añade la siguiente linea al LILO se desactiva el proceso de detección del parport:

```
parport=0
```

Con la intención de unificar los parámetros que necesita cada dispositivo que usa el puerto paralelo también se ha creado unas recomendaciones. Por ejemplo para cargar el gestor de impresoras en los puertos 0 y 2 que ha detectado el parport se utilizan el siguiente parámetro:

```
#insmod lp.o parport=0,2
```

Aunque en el LILO se ha de indicar añadiendo:

```
lp=parport0 lp=parport2
```

Algunos dispositivos permiten parámetros extra, por ejemplo como ahora mediante la negociación IEEE1284 se puede averiguar que tipo de dispositivo está conectado en cada puerto es posible realizar la siguiente entrada en el LILO:

```
lp=auto
```

Con la anterior entrada solo añadirá aquellos puertos que tengan conectada un impresora, dejando libres el resto. Se ha de tener cuidado con esta opción pues si cuando bota el ordenador la impresora está apagada no detecta nada, aunque este problema podria solucionarse si cuando se envia una escritura sobre un `/dev/lpX` y no existe impresora se intenta realizar un detección.

7.3.4. Interacción con Proc FileSystem

Desde el principio de la fusión de proyectos se observó que seria de agradecer la existencia de una interacción con el Proc Filesystem, ver 3.4 en la página 62 para una información más detallada del Proc Filesystem.

Al cargar el module `parport` se ha de crear una entrada en `/proc/parport`. Este directorio posee un subdirectorio por cada puerto paralelo que se encuentra, los nombres de los directorios empiezan por cero y se van incrementando cada vez que se detecta un puerto paralelo nuevo.

Dentro de cada subdirectorio detectado se crean tres ficheros, `hardware`, `irq` y `devices`. Por ejemplo si se detectan dos puertos paralelos automáticamente se crea la siguiente estructura de ficheros.

```
#find /proc/parport -print
/proc/parport
/proc/parport/1
/proc/parport/1/hardware
```

```
/proc/parport/1/devices
/proc/parport/1/irq
/proc/parport/0
/proc/parport/0/hardware
/proc/parport/0/devices
/proc/parport/0/irq
#
```

/proc/parport/n/hardware Almacena información sobre el hardware utilizado por el puerto paralelo. Muestra información de la dirección `base` donde reside, la `interrupción` que usa, el canal de `DMA` que ha detectado, los `modes` que soporta, el `modo` en el que está operando el puerto paralelo, el resultado del bench realizado, y por último el fabricante del chip que ha detectado. Un ejemplo típico podría ser:

```
#cat hardware
base:   0x378
irq:    5
dma:    -1
modes:  SPP,ECPEPP,ECPPS2
mode:   SPP
chip:   unknown
#
```

/proc/parport/n/irq Este fichero informa de la interrupción utilizada por el puerto paralelo. Permite que el usuario, solo root, modifique el valor de la interrupción, esto es útil pues en algunos puertos paralelos es imposible detectar la interrupción.

Por ejemplo para indicar que el puerto paralelo use la interrupción 7 simplemente se ha de realizar:

```
#cat irq
-1
#echo 7 >irq
#cat irq
7
#
```

/proc/parport/n/devices Muestra la lista de dispositivos que se han registrado para utilizar el puerto paralelo. Existe una linea por cada dispositivo, mientras que el dispositivo que se encuentra activo posee una más a principio. Si algún dispositivo se ha registrado como lurk al final imprime LURK.

Un ejemplo con una unidad ZIP realizando transferencias y una conexión mediante PLIP muestra el siguiente aspecto:

```
#cat devices
```

```
+ppa0
 eplip0 LURK
#
```

7.4. Implementación

Una novedad a consecuencia de la fusión es incorporar el proyecto parport dentro de un proyecto mayor, proporcionar al Linux la capacidad de aprovechar las especificaciones PnP de Microsoft. Por este motivo en los fuentes de Linux se ha creado un directorio de trabajo en `/drivers/pnp`, en este directorio residen todos los fuentes necesarios para operar con el puerto paralelo.

En la implementación se ha de considerar como se activa el módulo parport, así como los ficheros que han sido realizados y qué funcionalidades proporcionan.

7.4.1. Activación

Como ya se ha comentado el *parport* se comporta como un módulo. Con el parport se ha iniciado la creación de módulos que soporten la especificación PnP de Microsoft, por este motivo se ha optado por crear un nuevo directorio en `/drivers/pnp`. De igual manera se ha optado por crear una nueva entrada en la configuración del Kernel, de esta forma aparece una entrada para los dispositivos con *Plug and Play*.

Una vez seleccionada la entrada de PnP pregunta si queremos soportar el PnP, al igual que ocurre en el resto de opciones [8].

Para conseguir que aparezca la opción de configuración es necesario crear el fichero `/drivers/pnp/Config.in` con el siguiente contenido:

```
mainmenu\_option next\_comment
comment 'Plug and Play support'
bool 'Plug and Play support' CONFIG\_PNP
if [ "$CONFIG\_PNP" = "y" ]; then
 if [ "$CONFIG\_PNP\_PARPORT" != "n" ]; then
  bool 'Auto-probe devices' CONFIG\_PNP\_PARPORT\_AUTOPROBE
 fi
fi
endmenu
```

De esta forma cuando se ejecuta el `Makefile` del parport se podrá conocer cuál es la selección realizada. Se ha de recordar que los valores posibles son "y" para incluirlo en Kernel al tiempo de boot, "n" si no se requiere su compilación y "m" cuando opera como módulo.

7.4.2. Ficheros implicados

Se han creado cuatro ficheros en "C" con funcionalidades bien diferenciadas, estos ficheros se linkan para crear un único module que se llama **parport.o**. Los ficheros poseen la siguientes características:

[8] Para un descripción más detallada sobre como seleccionar módulos es conveniente leer la sección 3.2 en la página 43.

- **parport_init**, posee todo el código para la gestión del parport como module, y la funciones necesarias para detectar el puerto paralelo.
- **parport_probe**, funciones para identificar que tipo de dispositivo esta conectado al puerto paralelo, así como funciones para poder realizar las fases de negociación del estándar IEE1284 [9].
- **parport_procfs**, gestión del Proc Filesystem, proporciona información al sistema y permite que "root" pueda modificar la interrupción.
- **parprot_share**, alberga todas las APIs de que se especifican en el subsistema parport, exluyendo relacionadas con el estándar IEEE1284 que se albergan en parport_probe.

[9] Actualmente se está considerando la posibilidad de crear un nuevo fichero parport_ieee1284.c que albergaria todas las funciones relacionadas con el estándar de IEEE

7.5. Absorción de subsistemas

En el Kernel de Linux existen multitud de dispositivos que necesitan el puerto paralelo, por este motivo se ha proporcionado una API que permite una migración gradual.

En una primera fase se han pasado los dispositivos de impresoras *lp*, magneto óptico ZIP de Iomega *ppa* y la interconexión usando el módulo *PLIP*.

En cada driver que desee soportar el parport se han de realizar una seria de modificaciones, desde el método de conseguir los parámetros de inicialización, a la forma de registrar el puerto paralelo.

7.5.1. Parámetros de configuración

Los dispositivos que usan el módulo parport poseen unos parámetros unificados para simplificar la gestión. No obstante si necesitan algún parámetro adicional siempre pueden usarlo.

Los módulos poseen dos formas de conseguir parámetros, del LILO cuando bota la máquina o si se compila como module y se pasan como parámetros.

Parámetros usando LILO Los parámetros de LILO solo son válidos cuando el module se compila integrado con el Kernel, es decir cuando no es posible cargar y descargar el module.

Con la intención de homogeneizar la sintaxis se acostumbra a usar el nombre del módulo como cadena de entrada en la configuración del LILO [Almb] [Alma]. Por ejemplo el módulo *lp.o* acepta se configuraria en el LILO añadiendo la siguiente entrada en `/etc/lilo.conf`:

```
append = "lp=parport0"
```

La función que gestiona los parámetros del LILO acostumbra normalmente a llamarse, `X\_setup(char *str,int *ints)`, donde X corresponde al nombre del módulo. Esta función se llama tantas veces como parámetros existan.

El contenido de `str` corresponde al valor de la cadena pasada como parámetro, y `ints` es el vector que posee asignado la cadéna `str`. En el ejemplo anterior de configuración en el LILO, `str=''parport0''` e `ints[0]=0`.

Por ejemplo la función del módulo *lp* es:

```
void lp\_setup(char *str, int *ints)
{
if (!strncmp(str, "parport", 7)) {
  int n = simple\_strtoul(str+7, NULL, 10);
  if (parport\_ptr < LP\_NO)
    parport[parport\_ptr++] = n;
  else
    printk(KERN\_INFO "lp: too many ports, %s ignored.\n",str);
  } else if (!strcmp(str, "auto")) {
```

```
    parport[0] = -3;
  } else {
    if (ints[0] == 0 || ints[1] == 0) {
      /* disable driver on "lp=" or "lp=0" */
      parport[0] = -2;
    } else {
      printk("warning: 'lp=0x%x' is deprecated\n",ints[1]);
    }
  }
}
```

Para que se llame a esta función se ha de añadir una entrada en el `/init/main.c`, del tipo:

```
#ifdef CONFIG\_PRINTER
    { "lp=", lp\_setup },
#endif
```

Se ha de recordar que `CONFIG\_PRINTER` es el nombre que posee el módulo *lp* dentro del `/drivers/char/Config.in`. La entrada existente para el driver *lp* en Config.in también se ha de modificar pues ahora su existencia depende de que se compile el módulo parport. La nueva entrada en `/drivers/char/Config.in` quedaria:

```
dep\_tristate 'Printer' CONFIG\_PRINTER $CONFIG\_PNP\_PARPORT
```

Parámetros usando módulos La utilización de parámetros con los modules se ha simplificado mucho en las últimas versiones de Linux, actualmente solo se ha de declarar una variable que deseamos como parámetro de una forma especial.

```
static int parport[8] = {-1,};
MODULE\_PARM(parport, "1-8i");
```

El fragmento de código anterior declara una variable `parport` que será aceptada como parámetro y permitirá un vector de enteros entre 1 y 8. Por ejemplo la entrada:

```
#insmod lp.o parport=0,2
```

Dejaria la variable inicializada con parport[0] = 0, parport[1]=2 y el resto inicializados a -1.

7.5.2. Registrarse en el parport

Durante el proceso de inicialización se han de registrar los nuevos dispositivos dentro del parport.

Normalmente se recorre toda la lista de puertos paralelos existentes, y en cada puerto paralelo se intenta buscar la un dispositivo válido. Cuando encuentra un dispositivo se registra mediante `parport\_register\_device`. El ejemplo siguiente busca entre los puertos paralelos que soportan el modo PS2 y despues se registra.

```
static struct ppd *disp = NULL;
{
  int pos = 0;
  struct parport *pb = parport\_enumerate();

  while(pb){
    if( pn->modes & PARPORT\_MODE\_PS2 ){
      disp = parport\_register\_device(pb,name,NULL,lp\_wakeup,
                                      lp\_interrupt,NULL,
                                      "printer",NULL);
      if( !disp ){
        printk("Not enough memory\n");
        return -ENOMEM;
      }
      break;
    }
    pb = pb->next;
  }

  return 0;
}
```

El fragmento anterior registra un único dispositivo en el primer puerto paralelo que encuentra libre.

Al descargarse el módulo se ha de llamar a la función `parport\_unregister\_device`.

7.5.3. Adquisición del control del puerto

Una de las grandes novedades del parport es que permite disponer de varios dispositivos conectados en daisy chain, por lo que se ha de realizar una competencia por un recurso limitado entre distintos dispositivos.

Se ha de recordar que el device no posee el control del dispositivo, por lo que es necesario realizar un `parport\_claim` antes de realizar cualquier operación en el puerto paralelo. Una vez ha finalizado la utilización se ha de realizar un `parport\_release`.

```
int lp\_write(char *buffer, int size)
{
  int ret;

  if( parport\_claim(disp) ){
    sleep\_on(lp\_wait\_q);
  }

  ret = lp\_real\_write(buffer,size);
```

```
  parport\_release(disp);

  return ret;
}
```

Si el puerto está siendo usado por otro dispositivo no permitira realizar un `parport\_claim`, por lo que será necesario esperarse hasta que se llame a la función `lp\_wakeup` que se ha pasado como parámetro al registrar el dispositivo. Para poder despertar el flujo de ejecución que se ha dormido la función `lp\_wakeup` ha de poseer una funcionalidad similar a:

```
int lp\_wakeup(void *handle)
{
  if( parport\_claim(disp) )
    return 1;

  wake\_up(lp\_wait\_q);

  return 0;
}
```

7.6. Conclusiones

El nuevo subsistema de puerto paralelo ha sido creado mediante la colaboración de varios miembros voluntarios con distintos puntos de vista, estos voluntarios, distribuidos por todo el planeta, han utilizado Internet como sistema de comunicación. Como consecuencia posee una API ampliamente consensuada por la comunidad Linux.

El módulo parport está ampliamente optimizado, cuando se carga en kernel el espacio memoria requerido no alcanza los 28Kbytes, por lo que no supone una gran carga. Proporciona a los módulos una interfaz unificada, con capacidad de detección más avanzada de la que poseía cualquier módulo de Linux antes de implementar el parport, proporciona una detección de las características de cada módulo, una compartición del puerto paralelo y permite que los dispositivos conectados operen mientras están conectados en daisy chain.

Desde la versión 2.1.33 del Kernel de Linux se ha empezado a asimilar nuevos dispositivos, actualmente ya están asimilados el PLIP, LP y PPA [10].

Linus Torvalds ha fusionado el proyecto parport en la rama principal de Linux desde la versión 2.1.33, versión actual en fase de desarrollo. De esta forma ya se está distribuyendo a nivel mundial con todas las nuevas versiones de Linux, esta distribución masiva facilita la detección de posibles bugs o incompatibilidades con hardware sobre el que no se han realizado tests.

En el momento que se escribía esta memoria se estaba discutiendo una ampliación de la interfaz que posiblemente estaría operativa antes de la versión 2.1.50. Esta nueva interfaz proporcionaría funciones de entrada y salida para cada modo de funcionamiento del puerto paralelo, así como las funciones más usadas en la gestión de mascaras con los registros de control y status.

[10] Se posee una unidad de CDROM backplane y se ha contactado con Grant R. Guenther para que durante el mes de Julio se puede asimilar

Capítulo 8 ❑

Threaded Device Support

8.1. Introducción

Al comenzar a analizar la estructura de red en el Kernel de Linux se pudo observar que era aconsejable crear un módulo externo que podría ser utilizado por diversos dispositivos de red como un nuevo interfaz del subsistema de red.

Desde el principio se ha optado por usar los módulos de Linux, pues tal y como se indica en 3.3 ofrecen numerosas ventajas de desarrollo. En la sección 3.3.4 que aparece en la página 57 se puede encontrar la filosofía de diseño que se aplica a la hora de diseñar un nuevo módulo.

El nuevo módulo, principalmente soluciona dos problemas que se pueden plantear con algunos dispositivos de red:

- *Si el dispositivo de red es lento*, en este caso es necesario crear un thread o usar botton halves para no bloquear el kernel.

- *Si desconoce con antelación el tamaño del paquete que espera recibir*, entonces ha de preparar con antelación un buffer y despues copiarlo al buffer de red. [1]

Normalmente el kernel soluciona el problema de dispositivos lentos mediante la utilización de botton halves. Aunque existe un subsistema específico de reciente creación en Fujitsu AP1000 que utiliza threads en vez de botton halves. No obstante en máquinas multiprocesador es aconsejable utilizar threads en vez de botton-halves, mientras que en monoprocesadores no es tan importante.

8.2. Características deseadas

Se ha de destacar que en las últimas veriones de Linux el subsistema de red permite multithreading, con protección en las secciones críticas. De forma que el Kernel se encuentra optimizado para operar con threads, pues los botton halves no aprovechan este

[1] Aunque el driver conozca cual es el máximo teórico que puede llegar a alcanzar no se puede aprovechar un buffer previamente reservado.

aumento de la granularidad. Solo por este motivo ya seria aconsejable crear threads para que pudieran aprovechar esta característica.

Al analizar los drivers de red, se ha encontrado la existencia de una gran cantidad de módulos que utilizan los botton halves. Por este motivo en vez de implementar en cada módulo la gestión del thread o botton half, es aconsejable usar un módulo común que realice toda la gestión. Actualmente se duplica el código, por lo que esta mejora podría reducir el tamaño global del Kernel si varios módulos la adaptasen.

El otro problema que tiene que solucionar el TDS proviene de la gestión de buffer. Hasta el momento el Kernel no permite que el dispositivo de bajo nivel reserve los buffers por su cuenta y que despues los pase al subsistema de red. Si cuando recibe el paquete de información no conoce el tamaño de la información ha de copiar el contenido al nuevo buffer de red, con la consiguiente pérdida de prestaciones al realizar una copia [2].

En resumen las características que posee el nuevo módulo son:

- Facilitar la gestión de dispositivos de red lentos, con la creación de un thread [3].
- Permitir que el device de red utilice buffers previamente reservados para comunicarse con el subsistema de red.
- Facilitar la gestión de estadísticas [4].

El módulo TDS, Threaded Device Support, estructuralmente está entre los dispositivos de red (SNIP, EPLIP ...) y el subsistema de red, tal y como muestra la figura 8.1. No obstante es posible que en un futuro se integre con el subsistema de red, y se comporte como una extensión de la API que posee el subsistema de red.

El esquema muestra tres tipos de relaciones entre módulos:

Relación A La API estándar existente entre los dispositivos de red y el subsistema de red. Es una relación 1 a n puesto que el subsistema de red soporta simultaneamente múltiples dispositivos de red.

Relación B La relación entre el TDS y el subsistema de red, el TDS interacciona directamente con el subsistema de red, pero no existe la posibilidad de poseer varios tipos de TDS. Aunque posee varios parámetros de configuración.

Relación C La nueva API proporcionada por el TDS a cada uno de los dispositivos de red. Al igual que la relación A, se trata de una relación 1 a n puesto que el TDS soporta simultaneamente múltiples dispositivos de red.

[2] Se ha de observar que actualmente la limitación en ancho de banda del subsistema de red proviene del tiempo que pierde en copiar de memoria a memoria

[3] En [BG93] se aconseja mapear un procesador por mensaje. La implementación del TDS se comporta de forma similar pues cada thread solo puede procesar un mensaje, no obstante seria cuestión de analizar si aumentar el número de threads al máximo de mensajes simultaneos que puede utilizar cada driver siempre que sea inferior al total de CPU's existentes produce un aumento del throughput sin aumentar la latencia.

[4] Es posible que esta opción no sea bien recibida por la comunidad de Linux.

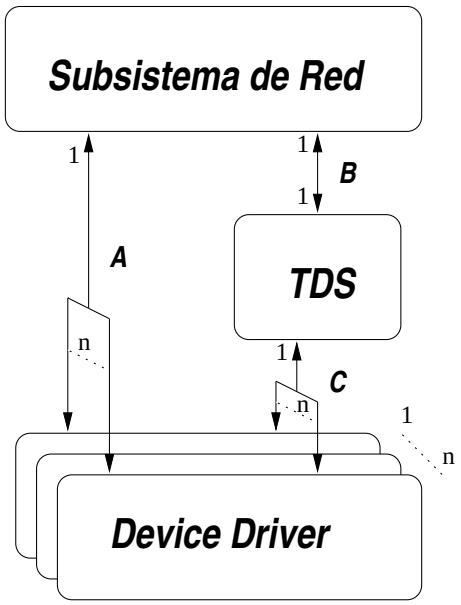

Figura 8.1.: Posicionamiento del TDS dentro de Linux

8.3. Diseño

En el diseño del TDS se ha intentado crear una API coherente con el resto del subsistema de red.. Partiendo de esta API, una vez sea aceptada por la comunidad Linux, podrá ser integrada dentro del subsistema de red mediante un proceso de propuestas de ampliación o modificación en alguna mailing list especializada en red.

Una vez que en la sección 8.1 ha sido aclarada la funcionalidad que ha de proporcionar, se han de empezar a definir las características de la implementación para que aparezca en las últimas versiones de Linux, actualmente la versión *Linux 2.1.40*.

Mientras que en la sección 8.3 se describe como opera funcionalmente, en el diseño del TDS se clarifica:

- El *diseño de las estructuras necesarias* para gestionar los Threads, y la extensión de la API del subsistema red.
- Se ha de incorporar la *especificación de una API* consistente con los módulos actuales, permitiendo que se pueda llegar a integrar en el subsistema de red.
- Permitir que funcionen buffers previamente creados.

8.3.1. Estructuras definidas

Antes de empezar con la especificación de la API es conveniente realizar una introducción a las dos estructuras principales que se han creado.

El TDS utiliza principalmente la estructura `TDS\_target`, donde almacena toda la información sobre la cola de comandos pendientes e información sobre el device de red. El dispositivo de red ha de reservar espacio para esta estructura, rellenarla y despues registrarla dentro del TDS.

Los campos que poseen la estructura `TDS\_target` son:

open: `int (*open) (struct TDS\_target *tar);`
Función que ha de proporcionar el driver de red, es la equivalente a la función `int open(struct device *dev)` que posee la estructura `device`, pero en lugar de pasar un puntero al device como parámetro pasa un puntero a la estructura TDS_target.

close: `int (*close) (struct TDS\_target *tar);`
Función inversa a `open`, ha de ser proporcionada por el driver de red, equivale a la función `int close(struct device *dev)` que posee la estructura `device`, pero en vez de pasar un puntero al device como parámetro pasa un puntero a la estructura TDS_target.

read: `int (*read)(struct TDS\_target *tar,char *buffer,int size);`
Los dispositivos de red que necesiten operar con threads para realizar las lecturas, han de proporcionar esta función. Cuando el device de red recibe un mensaje avisa al TDS, y el TDS se apunta como pendiente realizar una lectura. Una vez

se ha finalizado la interrupción activa un thread que llama a la función `read` del TDS_target para que realice la lectura. El TDS garantiza que las operaciones de lectura se realizen en el mismo orden con el que se han producido.

write: `int (*write)(struct TDS\_target *tar,char *buffer,int size);`
Esta función es llamada cuando el Kernel quiere enviar algún mensaje al dispositivo de red. Se ha de observar que cuando se utiliza el TDS no es posible usar la función que se utiliza con el resto de dispositivos de red (hard_start_xmit). Al igual que ocurre con el `read`, el TDS se encarga de encolar al final de las operaciones que posee como pendientes, opera como una FIFO compartida para las operaciones de lectura y escritura.

priv: `void *priv;`
Usado por el dispositivo de red para almacenar la información que considere oportuna [5].

dev: `struct device *dev;`
Apunta a la estructura device que también ha de registrar el dispositivo de red.

flags: `unsigned char flags;`
Características que posee el TDS en tiempo de creación, se pueden seleccionar tanto la forma de gestión de la recepción de mensajes como el método de enviarlos. Existen las siguientes opciones:

TDS_SEND_THREAD Un thread de Kernel se encarga de gestionar los paquetes que se envían del subsistema de red al driver de red.

TDS_SEND_BOTTONHALF Los paquetes que se envían se gestionan mediante botton halves.

TDS_SEND_SHORTCUT El mensaje se enviar directamente en espacio de kernel. Esta opción solo es útil cuando el dispositivo envía paquetes muy rápidamente.

TDS_RECV_THREAD Un thread de Kernel se encarga de gestionar los paquetes que se reciben en el driver de red y se tienen que pasar al subsistema de red.

TDS_RECV_BOTTONHALF Los paquetes que se reciben se gestionan mediante botton halves.

TDS_RECV_SHORTCUT El mensaje se recibe directamente en espacio de kernel. Esta opción solo es útil cuando el dispositivo envía paquetes muy rápidamente.

wait_for: `int wait\_for;`
Tiempo de espera que ha de realizar el TDS entre comandos de lectura o escritura cuando se ha producido un error. El tiempo se indica en jiffies, si el valor de `wait\_for` es cero se espera 10uS antes de volver a ejecutar el comando.

[5] Ahora el dispositivo de red no puede utilizar la variable priv de device para almacenar información privada, pues en priv se apunta a TDS_target

stats: `struct net\_device\_stats stats;`
Estadísticas que gestiona el TDS, aunque el dispositivo de red puede operar con ellas.

pool: `struct tds\_cmd\_pool pool;`
Junto con stats esta es la única estructura que gestiona el TDS y sobre la que el dispositivo de red no ha de inicializar. Se utiliza para gestionar la FIFO de mensajes pendientes que posee el TDS.

8.3.2. Especificación de la API

El TDS exporta un grupo muy reducido de funciones, en realidad solo se trata de tres funciones nuevas. Estas funciones pueden ser llamadas por los dispositivos de red y en cierta manera parece una extensión del subsistema de red.

No obstante como el TDS ha de poseer la capacidad de llamar a múltiples dispositivos de red, también se especifica como API las funciones que ha de proporcionar cada uno de los dispositivos de red.

int tds_alloc(struct TDS_target *tar); Cada dispositivo que desea utilizar el TDS se ha de registrar mediante la función `tds\_alloc`.

Antes de llamar a `tds\_alloc` se ha de rellenar la estructura `TDS\_target`. En caso de producirse un error en la inicialización `tds\_alloc` retorna un valor negativo.

Se ha de recordar que `tds\_alloc` llama a `register\_netdev` por lo que los dispositivos que usen el TDS no han de llamarla para registrar un nuevo dispositivo.

void tds_free(struct TDS_target *tar); Función inversa a `tds\_alloc` libera todos los recursos y después llama `unregister\_netdev`.

int tds_prepare_recv(struct TDS_target *tar, unchar * buffer, int size, ushort proto); Cada vez que el dispositivo de red recibe un nuevo mensaje ha de llamar a `tds\_prepare\_recv` [6]

Si es imposible encolar el mensaje contabiliza que se ha producido un `overrun` y retorna cierto. En caso que pueda procesar el comando de lectura retorna falso.

Existen cuatro parámetros para llamar a `tds\_prepare\_recv`:

TDS_target El dispositivo de red ha de proporcionar el puntero a la estructura TDS_target que utilizó en `tds\_alloc`.

buffer Buffer a los datos que ha leído, este campo es opcional, pudiendo ser un puntero a un buffer o NULL. Dependiendo de si es NULL o apunta a un buffer el TDS posee un comportamiento claramente diferenciado.

[6] Esta función es bastante similar a `netif_rx` pues se encarga de notificar al kernel que ha llegado un nuevo mensaje, no obstante permite muchas opciones que son imposibles con `netif_rx`

NULL Cuando NO se proporciona un buffer el TDS considera que posee pendiente una lectura. Programa la operación de lectura y más tarde realiza una llamada a la función `read` que ha programado el dispositivo de red. Este caso es especialmente útil para aquellos dispositivos de red que operan lentamente y es aconsejable que el proceso de lectura no se realice mientras ocurre la interrupción.

buffer Si el dispositivo de red ya posee el mensaje recibido, en buffer se encuentran los datos del mensaje, en la sección 8.4 de la página 148 se describe como se gestionan los buffers de red en el TDS. Cuando el dispositivo de red proporciona el buffer el TDS NO llama a la función `read` pues ya posee toda la información necesaria.

size Tamaño del paquete de datos que se ha recibido o se espera recibir.

proto El dispositivo de red ha de proporcionar información sobre el protocolo que identifica el paquete de llegada [7].

[7]Normalmente esta información la encapsula el nodo remoto en la cabecera de nivel de enlace

8.4. Modificaciones en el subsistema de red

Para poder optimizar la gestión de paquetes de los niveles inferiores, es necesario realizar unas pequeñas modificaciones (o mejor dicho, adiciones) en el sistema central de red del kernel. Estos cambios son efectuados para que los dispositivos que ya han recibido la totalidad del mensaje en el momento de la comunicación al TDS, no tengan ninguna penalización de copia para pasar los datos en el formato o memoria adecuada para el tratamiento en los niveles de red de kernel.

El mecanismo que utiliza el sistema central de red del Linux, para gestionar los diferentes paquetes que envía o recibe mediante dispositivos de red, es la estructura `sk\_buff`. Esta estructura tiene la información del paquete, como el destino, origen, tipo... además de los punteros a los datos asociados.

La única función que provee el núcleo para poder crear estos `sk\_buff`'s, es `alloc\_skb()`, que a la vez solo se utiliza llamándose directamente por la función `dev\_alloc\_skb()`. Por lo tanto, a efectos prácticos, si se quiere crear un nuevo *skb*, se debe utilizar la función `dev\_alloc\_skb()` pasándole como parámetro la longitud de memoria total a reservar para los datos, y la prioridad de memoria requerida.

La función `dev\_alloc\_skb()` junto con la `alloc\_skb()` devuelven los siguientes datos, una vez llamadas:

- Reservan memoria para la longitud requerida más el tamaño de una estructura `sk\_buff`. Se hace en un espacio continuo, donde primero están los datos y seguidamente el *skb*.
- Inicializan los campos de la estructura de `sk\_buff` reservada, para que los datos apunten a la memoria también reservada.
- Dejan un espacio (*gap*) de 16 bytes al inicio de los datos, para posibles ampliaciones, tal como se muestra en la figura 8.2.

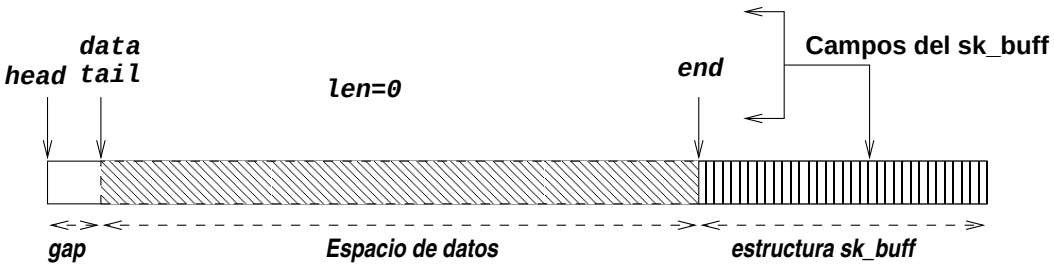

Figura 8.2.: Estructura del espacio de memoria devuelto por *dev_alloc_skb()*.

Como se ve, al crear un *skb*, la misma función reserva la memoria y crea una estructura común para su manejo. Con este mecanismo, no se puede crear un *skb* pasando como parámetro el área de memoria para los datos.

Por esta razón, se han creado unas nuevas funciones, llamadas `NEW\_dev\_alloc\_skb()` y `NEW\_alloc\_skb()` que son las equivalentes a las anteriormente comentadas, y que permiten pasar el área de memoria a utilizar para los datos y *skb*.

La función que se usa directamente es `NEW\_dev\_alloc\_skb()` la cual acepta un segundo parámetro como puntero al inicio de la memoria a utilizar. Esta memoria debe estar ya reservada, y debe tener una longitud mayor o igual a la suma de los datos, la estructura `sk\_buff` y el *gap*. Con este nuevo parámetro, se llama a la función `NEW\_alloc\_skb()` que no reserva la memoria, o se llama a la original `alloc\_skb()` en caso de que este sea nulo.

Vistos los nuevos mecanismos, ahora un nivel inferior al TDS, solo debe tener pasar como paquete recibido una área de memoria que contenga el inicio de los datos en un *offset* del tamaño del *gap*(16 bytes), y que le sobre un espacio de memoria de tamaño de un `sk\_buff` al finalizar la parte de datos.

Así pues, un ejemplo completo desde el nivel más bajo hasta llegar al TDS y enviándolo al sistema de red del linux, seguiría la siguiente evolución. El ejemplo se basa en que pasos sucederán con un paquete recibido de 100 bytes:

1. El nivel más bajo reserva un espacio de memoria de tamaño igual a la suma de $gap + datos + skb$.Figura 8.3.

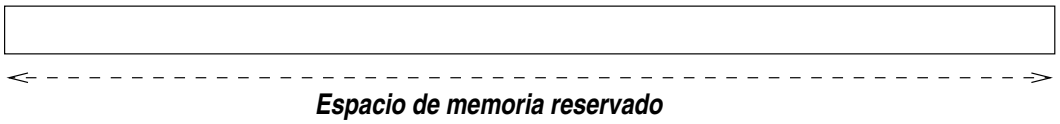

Figura 8.3.: Espacio total reservado para datos más *skb*

2. Al recibir el paquete, copia los datos (incluida la cabecera) sobre el buffer reservado, con un offset del *gap*. Es decir, deja un pequeño espacio al inicio del buffer.Figura 8.4.

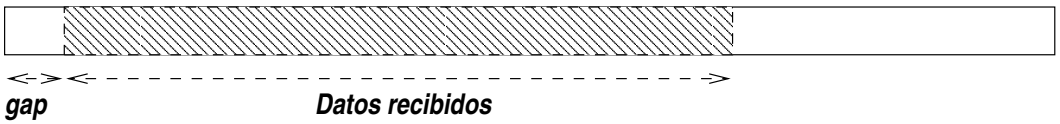

Figura 8.4.: Ubicación de los datos dentro del espacio reservado

3. El paquete se pasará hacia los niveles superiores hasta llegar al TDS mediante la función (`tds\_prepare\_recv()`).

4. El TDS llamará a la nueva función `NEW\_dev\_alloc\_skb()` pasándole como parámetro el buffer ya reservado. Con este paso quedará un *skb* inicializado correctamente como hace por defecto el núcleo, y con la memoria que ya se tenia reservada.Figura 8.5.

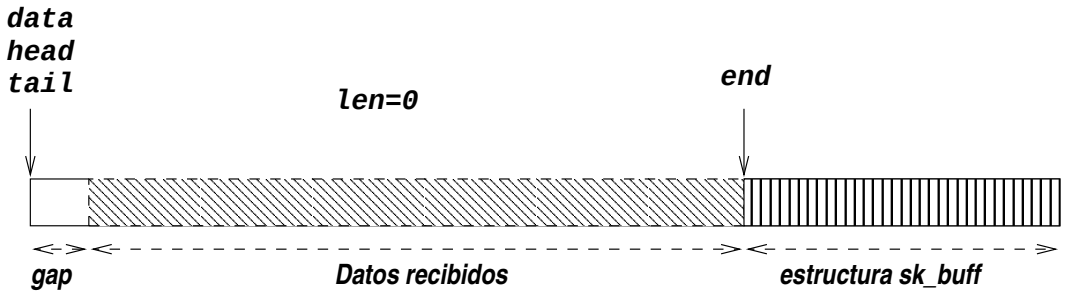

Figura 8.5.: Estado de la memoria después de la función *NEW_alloc_skb()*.

5. Después el TDS marcará el *skb* conforme tiene 100 bytes de datos (función `skb\_put()`), y marcará el inicio de los datos válidos saltándose la cabecera del dispositivo (función `skb\_pull()`).Figura 8.6.

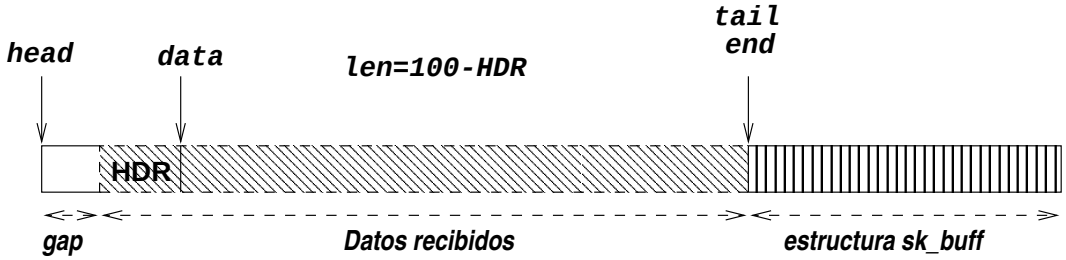

Figura 8.6.: Aspecto de un paquete+*sk_buff* preparado para el kernel de red.

6. Finalmente, llamará al nivel superior con el *skb* perfectamente creado y rellenado como se espera.

   Así pues, utilizando este sistema, se optimiza en gran manera, la gestión de memoria de buffers para transmitir o recibir entre dispositivos de red, ya que utilizando el método convencional, haría falta volver a copiar toda la parte de datos en un nuevo `sk\_buff` y reindexar los valores de la estructura, lo cual es costoso, sobretodo con tamaños de paquetes grandes.

8.5. Implementación

En la implementación se ha de considerar como se activa el módulo TDS, así como los ficheros que han sido realizados y qué funcionalidades proporcionan.

8.5.1. Activación

Como ya se ha comentado el TDS se comporta como un módulo. La activación del módulo de TDS se realiza mediante la activación de drivers experimentales en la

configuración y entonces cuando se configuran el entorno de red pregunta si se quiere compilar el *TDS Threaded Device Support (EXPERIMENTAL)* [8]

Para conseguir que aparezca la opción de configuración es necesario añadir en `/net/Config.in` una entrada como la siguiente, al final del fichero:

```
tristate 'TDS Threaded Device Support (EXPERIMENTAL)' CONFIG\_TDS
```

De esta forma cuando se ejecuta el `Makefile` del TDS se podrá conocer cuál es la selección realizada. Se ha de recordar que los valores posibles son "y" para incluirlo en Kernel al tiempo de boot, "n" si no se requiere su compilación y "m" cuando opera como módulo.

8.5.2. Ficheros implicados

Los ficheros usados en la creación del módulo TDS se encuentran en `/net/tds`. Por un lado encontramos el `Makefile` que dependiendo de las opciones de configuración seleccionará una u otra forma de compilar cada fichero.

Además del `Makefile` existen tres ficheros en "C" y un include que agrupan funcionalidades diferenciadas pero que interaccionan dentro del mismo módulo que se llama **tds.o**.

- `tds.h` posee la definición de las estructuras que exporta el TDS (`TDS\_target`) y los prototipos de funciones que exporta el módulo TDS. Por estos motivos este include ha de ser usado por todos los subsistemas de red que necesiten utilizar el TDS.

- `tds\_interface.c` gestiona el módulo, y tambien incorpora todas las funciones exportadas a otros subsistemas (`tds\_open`, `tds\_stop`, `tds\_send`, `tds\_stats`, `tds\_prepare\_recv`, `tds\_alloc` y `tds\_free`.

- `tds\_dev.c` posee todas las funciones de soporte que utiliza el thread o botton half para comunicarse con los dispositivos de red. Se ha de destacar las funciones `tds\_real\_send`, `tds\_recv\_with\_buffer` y `tds\_real\_recv`.

- `tds\_daemon.c` fichero que incorpora todas las funciones necesarias en la creación y gestión del thread, así como la gestión de la FIFO de comandos pendientes que ejecuta el thread. Se ha de destacar las funciones `tds\_add\_command`, `do\_command`, `tds\_daemon` y el cuerpo del thread que se llama mediante `tds\_daemon\_init`.

[8] Para un descripción más detallada sobre como seleccionar módulos es conveniente leer la sección 3.2 en la página 43.

8.6. Funcionamiento

Tal y como se ha comentado en 8.5.2 existen tres ficheros en "C" que permiten gestionar el TDS.

Con intención de facilitar como interaccionan las funciones de estos tres módulos a continuación se describe como funcionaria una petición de lectura con buffer, otra sin buffer y una escritura a un dispositivo cualquiera.

8.6.1. Enviar Mensaje

A continuación se documenta que funciones intervienen en el proceso de enviar un mensaje, de esta forma es fácil hacerse una de cual es el coste y cuales son las ventajas que proporciona usar el módulo TDS en vez de usar directamente el subsistema de red.

Cuando el nivel de red quiere enviar un mensaje llama a la función `tds\_send`, esta función ha sido registrada por el TDS en la estructura `device->hard\_start\_xmit` asociada a cada dispositivo.

En `tds\_send` se verifica que se trate de un mensaje correcto [9], inmediatamente despues llama a la función `tds\_add\_command` que encola como tarea pendiente del thread el mensaje a enviar.

Una vez se concede CPU al thread [10], se llama a la función `do\_command` que se encuentra en `tds\_daemon.c`.

Esta función busca en la FIFO cual es el siguiente comando que necesita ejecutar, en este caso un `TDS\_CMD\_SEND`, y llama a la función asociada.

El proceso final se realiza llamando a la función `tds\_real\_send` que reside en `tds\_dev.c`, llama a la función que ha proporcionado el driver de red en la estructura `TDS\_target`.

Una vez se ha enviado el mensaje contabiliza las estadísticas y elimina el buffer usado.

La figura 8.7 muestra de una forma gráfica las funciones que intervienen en enviar un mensaje. También incorpora información de los tiempos que intervienen en cada mensaje que se envía. Ha sido imposible calcular los tiempos de forma empírica por lo que se proporciona una aproximación del número de instrucciones assembler que necesita ejecutar en cada fase.

- **T1** El tiempo usado en verificar la información que se recibe, este tiempo no es nuevo pues también es necesario sin usar el TDS. Suele ocupar unas 56 instrucciones.

- **T2** Tiempo gastado en encolar la petición de un nuevo mensaje y activar el thread. Este tiempo no existía con antelación pues supone una tarea extra. Este proceso no llega a 134 instrucciones assembler.

[9] Ocasionalmente el subsistema de red envía mensajes vacíos que lo único que en realidad significan es un reset de cierto subsistemas `dev_tint`

[10] Con tal de minimizar la penalización en la latencia el thread es del tipo real time, por lo que una vez necesita CPU inmediatamente se le concede.

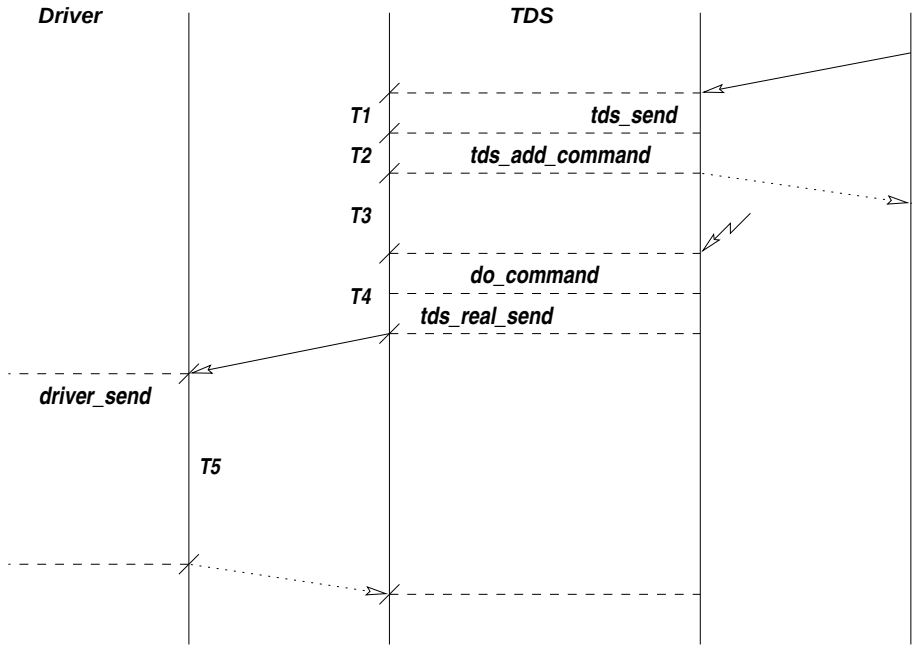

Figura 8.7.: Transmisión de un mensaje

- **T3** Tiempo de espera entre que se finaliza el comando `tds\_send` y se activa el thread. Este tiempo no se puede calcular pues depende de la carga de la máquina y otros factores, aunque en condiciones normales oscilará entre 1uS y 2uS.

- **T4** Tiempo gastado en buscar en las listas el comando a ejecutar y llamar a la función almacenada en `TDS\_target->write`, supone aproximadamente 164 instrucciones.

- **T5** Tiempo que gasta el driver en enviar la información, este tiempo depende totalmente del dispositivo existente.

El tiempo extra que suponen todas las fases extras no llega a suponer 4uS, considerando que actualmente estas latencias son muy reducidas y que a medida que se aumenta las prestaciones de los procesadores este tiempo se reducirá, es más que aceptable. La transmisión más rápida con placas fast ethernet [11] posee una latencia de envio de 90uS por lo que el overhead proporcionado por el TDS no alcanza un 4.4 %.

8.6.2. Recibir mensaje

A continuación se documenta que funciones intervienen en el proceso de recibir un mensaje. Si se analiza conjuntamente con el coste de enviar un mensaje, es fácil hacerse

[11] Estos tiempos se basan en una máquina UltraSparc por lo que es factible que tardase menos de 4uS en realizar todo el proceso

una de cual es el coste y cuales son las ventajas que proporciona usar el módulo TDS en vez de usar directamente el subsistema de red.

Cuando el driver de red recibe un mensaje llama a la función `tds\_prepare\_recv`, que automáticamente llama a `tds\_add\_command` para marcar como pendiente el nuevo mensaje.

Una vez se concede CPU al thread, se llama a la función `do\_command` que se encuentra en `tds\_daemon.c`.

Esta función busca en la FIFO cual es el siguiente comando que necesita ejecutar, en este caso un `TDS\_CMD\_RECV`, y dependiendo si existe un buffer llama a `tds\_recv\_with\_buffer` o `tds\_real\_recv`.

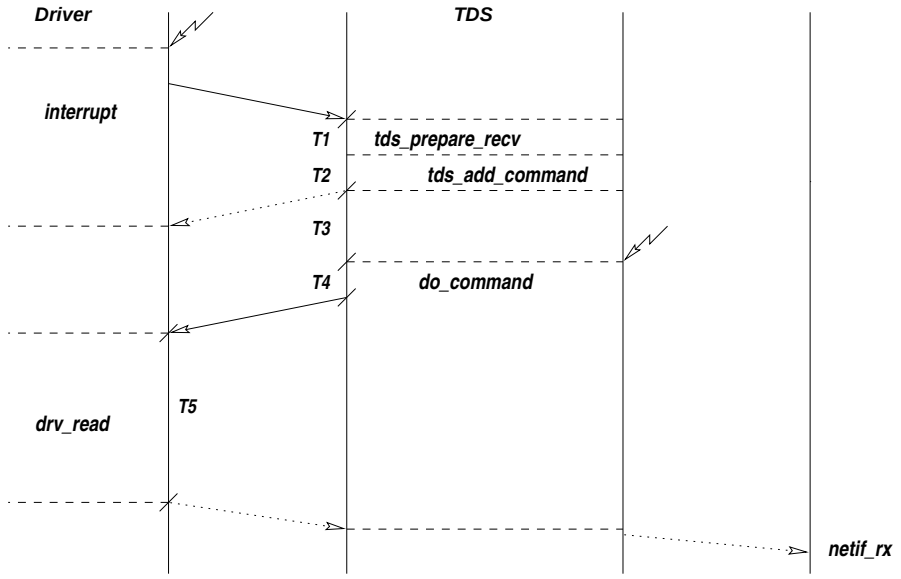

Figura 8.8.: Recepción de un mensaje

En caso que se llame a `tds\_recv\_with\_buffer` se realiza una llamada a `netif\_rx` para que envíe el mensaje al subsistema de red.

Si no se proporciona buffer entonces se llama a la función `TDS\_target->read` para que el driver de red realice el proceso de lectura. Esta opción es especialmente útil en dispositivos lentos.

Una vez se ha recibido el mensaje contabiliza las estadísticas y elimina el buffer usado.

La figura 8.8 muestra de una forma gráfica las funciones que intervienen en enviar un mensaje. También incorpora información de los tiempos que intervienen en cada mensaje que se envía. Al igual que ocurre con los tiempos de enviar un mensaje, ha sido imposible calcular los tiempos de forma empírica por lo que se proporciona una aproximación del número de instrucciones assembler que necesita ejecutar en cada fase.

- **T1** El tiempo usado en verificar la información que se recibe, este tiempo no es

nuevo pues también es necesario sin usar el TDS. Suele ocupar unas 56 instrucciones.

- **T2** Tiempo gastado en encolar la petición de un nuevo mensaje y activar el thread. Este tiempo no existía con antelación pues supone un aumento de la latencia. Este proceso no llega a 134 instrucciones assembler.

- **T3** Tiempo de espera entre que se finaliza el comando `tds\_send` y se activa el thread. Este tiempo no se puede calcular pues depende de la carga de la máquina y otros factores, aunque en condiciones normales oscilará entre 1uS y 2uS.

- **T4** Tiempo gastado en buscar en las listas el comando a ejecutar y llamar a la función almacenada en `TDS\_target->read`, supone aproximadamente 164 instrucciones.

- **T5** Tiempo que gasta el driver en recibir la información, este tiempo depende totalmente del dispositivo existente y no varía por usar o no el TDS.

Al igual que ocurre con el TDS, el tiempo extra que suponen todas las fases extras no llega a suponer 4uS.

Se ha de destacar que la latencia usando el *loopback*, (teóricamente la latencia mínima pues se trata de la máquina local), está en 190uS por lo que un aumento de 8uS que supondría un round trip representa una perdida de un 4 % en la latencia. No obstante esta perdida de latencia que se soluciona en dispositivos más rápidos usando como flag shortcut en vez de thread o botton half. Al registrarse en el TDS se usa la nueva opción que evita llamar al thread y realiza la llamada directamente.

8.7. Conclusiones

El módulo TDS permite una reducción del código en cada uno de los dispositivos de red, simplificando el desarrollo en los dispositivos de red lentos, así mismo no disminuye las prestaciones en latencia y aumenta a capacidad de utilización del Kernel pues se ejecuta como un thread que no realiza un `spin\_lock` al Kernel.

Al modificar el subsistema de buffers del Kernel permite que dispositivos como el SCSI puedan programar un buffer con antelación a la recepción del mensaje. De esta forma se evita tener que realizar una copia con el consiguiente aumento en el throughput. Técnicamente este sistema también permite reducir las latencias, no obstante no se ha podido realizar benchmarks con dispositivos una latencia suficientemente baja para poderlo demostrar empíricamente.

Tal y como se ha comentado con anterioridad el TDS provoca un aumento teórico del 4 % en la latencia si existiera un dispositivo con latencia 0. No obstante esta latencia que se produciría con dispositivos muy rápidos se evita creando un cortocircuito en los comandos de enviar y recibir, no obstante en este caso se pierde la ventaja de desbloquear al kernel durante la recepción de un mensaje.

Capítulo 9 ❑

EPLIP

El módulo EPLIP proporciona un nuevo driver de red que permite la interconexión de dos ordenadores usando el puerto paralelo como sistema de transmisión. A diferencia del PLIP opera con los nuevos modos del puerto paralelo, proporcionando unas prestaciones superiores.

9.1. Introducción

Las siglas EPLIP provienen de *Enhanced Parallel IP*, es un driver de red que permite conectar dos ordenadores mediante un cable paralelo usando el modo ECP.

Desde las primeras versiones de Linux existe el módulo **PLIP** que permite la interconexión de dos ordenadores usando el puerto paralelo. No obstante cuando se diseño el PLIP no existían los nuevos modos de puerto paralelo EPP y ECP, en el capítulo 4 de la página 88 se describen ampliamente los nuevos modos del puerto paralelo. Tanto la señalización, como el cable utilizado en la interconexión, del PLIP lo hacen incompatible con el modo EPP o el modo ECP. Por este motivo es necesario crear un nuevo driver de red.

El módulo EPLIP reside en el mismo directorio que el PLIP, `/drivers/net`. Al igual que ocurre en las últimas versiones de Kernel con el PLIP el EPLIP utiliza el parport en la gestión del puerto paralelo.

9.2. Características deseadas

Tal y como ya se ha dicho, el módulo de EPLIP, se encargará de la gestión del puerto paralelo para interconectar dos ordenadores.

Utiliza el módulo TDS, descrito en el capítulo 8, pues al tratarse de un dispositivo lento es muy útil aprovechar la gestión de colas con threads que facilita el TDS.

El concepto de funcionamiento del módulo es muy sencillo, el TDS se comporta como cualquier driver de red de Linux, pero posee unas características especiales.

- A diferencia de la mayoría de módulos no gestiona directamente el hardware pues ha de usar el parport en la gestión del puerto paralelo.

- En vez de implementar directamente una gestión mediante threads o botton halves utiliza el TDS que está especialmente diseñado para ocasiones como esta.

El EPLIP ha de cumplir las siguientes especificaciones de diseño:

- Ha de utilizar el parport para gestionar el puerto paralelo.
- Permitir la interconexión de dos ordenadores usando un cable paralelo compatible con el estándar IEEE1284 [Cor93b].
- Ha de soportar el modo ECP en la transmisión de información.
- Usar el módulo TDS como API con el subsistema de red.

9.3. Diseño

A la hora de diseñar el EPLIP de forma que se cumplan las característias deseadas especificadas en 9.2 es necesario que el EPLIP utilice el parport y el tds.

Se entiende más claramente como funciona el EPLIP mediante la especificación de las estructuras que necesita, como gestiona los mensajes, y una explicación de como se inicializa el EPLIP.

9.3.1. Estructuras

Hay diferentes estructuras y variables que juntas permiten al nivel de EPLIP manejar los diferentes módulos que pueden haber bajo su control.

Cada puerto paralelo que posee una interconexión entre dos ordenadores y se desea que opere con el EPLIP ha de poseer tres estructuras, `struct device`, `struct TDS\_target` y `struct eplip\_net`. Las dos primeras estructuras se explican en 3.5 y 8.1 respectivamente, mientras que `struct eplip\_net` es específica del EPLIP.

struct eplip_net Por cada dispositivo EPLIP que se registra en el subsistema de red ha de existir esta estructura, donde posee los siguientes campos:

pardev Es del tipo `struct ppd *pardev`, posee a la estructura retornada por la función `parport\_register\_device` al buscar un puerto paralelo con el que operar.

opened `atomic\_t opened` contador para conocer si sobre el dispositivo se ha realizado un `ifconfig`.

inuse `int inuse` cierto mientras se está realizando alguna operación de transmisión o recepción con el puerto paralelo mediante el EPLIP.

port_owner `int port\_owner` con el parport es necesario ceder el control a otros dispositivos que se encuentran en daisy chain, mediante este flag es posible saber si se posee el control del bus o si es necesario pedirlo antes de realizar una transmisión.

should_relinquish `int should\_relinquish` cuando el parport reclama el control del puerto paralelo pero no puede cederlo pues se está realizando una transmisión, activa esta variable para ceder el control una vez se finalize la transmisión en curso.

9.3.1.1. Control de buffers

En el diseño de las cabeceras de nivel de enlace a usar existió dos opciones, usar las mismas cabeceras que utiliza el PLIP o crear unas nuevas.

El PLIP utiliza tramas ethernet (14Bytes) como cabecera de nivel de enlace, esta opción facilita el desarroyo del módulo. No obstante aumenta la latencia pues siempre se añade una cabecera de 14bytes.

En el EPLIP se ha optado por crear una cabecera de nivel de enlace propia, solo se usan 2bytes para el tamaño del paquete y 2bytes para la identificación del protocolo que encapsula.

La estructura que tiene una cabecera de EPLIP para los paquetes transmitidos es la siguiente:

```
struct eplip\_hdr{
  \_\_u16 protocol;
  \_\_u16 len;
};
```

Donde cada uno de los campos corresponde a:

protocol Es el numero de protocolo que indica de que tipo es el paquete que viene encapsulado en los datos transmitidos por el puerto paralelo. Sigue la misma relación que el campo de protocolo que utiliza el encapsulamiento de tramas Ethernet (ver capítulo 2), es decir 16bits.

len Este campo también de 16bits, indica la longitud del paquete transmitido. Con 16bits sólo se pueden transmitir paquetes de hasta 65536 bytes, pero debido a que los niveles del TCP/IP lo limitan a menos, se ha considerado un numero máximo adecuado.

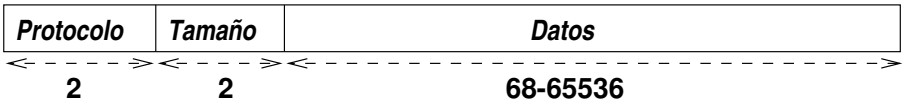

Figura 9.1.: Formato de un paquete EPLIP con su cabecera.

Una opción analizada consistia en utilizar señalización fuera de banda para la cabecera del EPLIP, no obstante como esta opción no proporciona ninguna ventaja se ha optado por transmitir todo por el campo de datos.

Debido a que la transmisión en el puerto paralelo es punto a punto, no es necesario que los paquetes tengan una dirección origen y destino para saber a quienes implica la transmisión.

9.3.2. Inicialización

Es en la inicialización del sistema donde se preparan y empiezan a utilizar todas las estructuras y variables anteriormente detalladas.

Una vez que el modulo se ha inicializado, deben hacerse las operaciones pertinentes para que se configure adecuadamente con el TDS. Siguiendo la filosofía del diseño, los modules advierten de su existencia o capacidad a los módulos superiores.

Se ha de indicar que el EPLIP permite un paramétro de configuración, al cargarse el módulo es posible indicar una lista de puertos paralelos sobre los que ha de intentar registarse el EPLIP. Esta opción se realiza mediante el parámetro `parport`. Por ejemplo cargar el módulo parport indicando que lo intente sobre los puertos 1 y 2 se realiza de la siguiente manera:

```
#insmod parport.o parport=1,2
```

Como se ha comentado con anterioridad el EPLIP utiliza los módulos TDS y PARPORT, por lo que es necesario que se encuentren cargados en espacio de kernel con anterioridad.

Para inicializar el TDS se ha de proporcionar la siguiente información:

- En la estructura de `device` (referente al dispositivo de red de alto nivel) sólo se indicara el nombre y función de inicialización `eplip\_setup`.
- En la estructura de `TDS\_target` se indicarán las funciones genéricas `eplip\_open()`, `eplip\_close()`, `eplip\_write()` y `eplip\_read()`.
- En la estructura de `eplip\_net` se completaran todos los campos, con la información que tenemos en los parámetros de la función.
- Reserva un `pardev` mediante la función `parport\_register\_device`, a esta función se le ha de proporcionar las funciones de control del puerto paralelo `eplip\_wakeup` y `eplip\_preempt`, también se ha de proporcionar el gestor de la interrupción `eplip\_interrupt`.

A la vez, estas tres estructuras juntas, se referencian entre si por medio del campo `priv`, de manera que `device` tiene un puntero hacia `TDS\_target`. Esta otra tiene un puntero hacia `eplip\_net` y esta ultima tiene otro hacia `TDS\_target`.

Una vez que las funciones `tds\_alloc()` y `parport\_register\_device` se han ejecutado correctamente, los tres módulos implicados tendrán constancia de que hay una tarjeta de red más en el sistema.

La inicialización del nivel superior de red, se completará cuando éste ejecute la función `eplip\_setup()`. Esta función completará todas las definiciones necesarias referentes al nivel de red del kernel.

Completados todos los pasos, se tendrá un nuevo dispositivo de red registrado en el sistema. Este no será operativo, hasta que se utilicen las funciones típicas del sistema para su configuración como *ifconfig*, lo que provocará la ejecución de algunas de las funciones programadas como `eplip\_open()`.

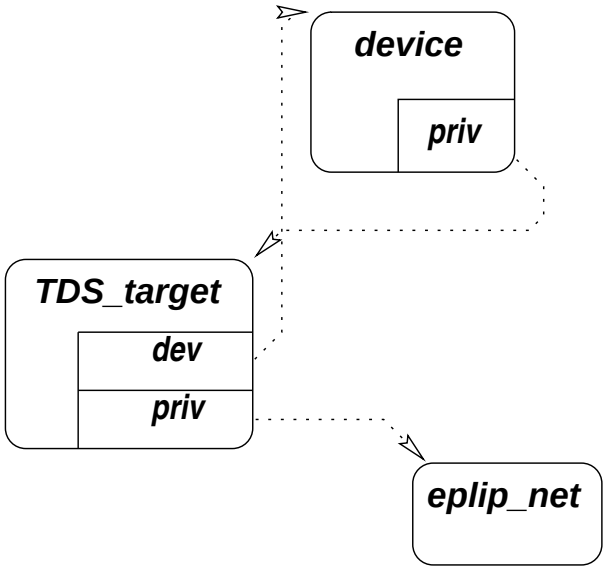

Figura 9.2.: Relaciones entre TDS, device y eplip_net

9.4. Funcionamiento

Una vez el proceso de inicialización y registro esta completo, el nuevo driver EPLIP está identificado en el sistema como un nuevo adaptador de red, y por lo tanto para empezar a utilizarla se debe configurar como cualquier otro dispositivo de red.

También cabe notar que si no se utilizara todo el stack TCP/IP con esta nueva tarjeta, no haría falta utilizar las herramientas del sistema para su configuración, sino que se deberían hacer las llamadas directamente a los módulos, ya sea mediante *system calls* o mediante un *filesystem device* a tal efecto.

Con todo, la explicación se basará en el uso sobre TCP/IP.

9.4.1. Activación del dispositivo

Para activar cualquier dispositivo de red, lo debemos configurar por ejemplo con la utilidad *ifconfig* de sistema. Esta activará una serie de llamadas del kernel y módulos intermedios, hasta que la función `eplip\_open()` será llamada, indicando que el nivel superior quiere empezar a utilizar el puerto paralelo como sistema de interconexión para transmisiones de red.

La función `eplip\_open()` hará las siguientes acciones para preparar el dispositivo de bajo nivel para la transmisión.

- Comprobación de posibles errores de parámetros.
- Marcará el dispositivo `eplip\_net` como en uso.
- Finalmente si no ha habido ningún error, efectuará una llamada a la función `parport\_claim()` con la intención de reclamar el control del puerto paralelo.
- Activa las interrupciones del puerto paralelo, de esta forma puede ser interrumpido cuando la máquina remota quiere enviar un mensaje.

Al llamar a `tds\_alloc` se han de pasar rellenados los campos de `priv`, `name` y `init` de la estructura device.

El TDS llamará a `register\_netdev` que a su vez llamará a la función que se ha definido en `dev->init`. En el EPLIP está función corresponde a `eplip\_setup`. Esta función debe completar todos los campos de la estructura `device` que sean necesarios para el tipo de dispositivo que vayamos a manejar.

```
int eplip\_setup(struct device *dev)
{
  dev->change\_mtu         = eplip\_change\_mtu;
  dev->hard\_header        = eplip\_hard\_header;
  dev->rebuild\_header     = eplip\_rebuild\_header;
  dev->hard\_header\_cache  = eplip\_header\_cache;
  dev->header\_cache\_update= eplip\_header\_cache\_update;
```

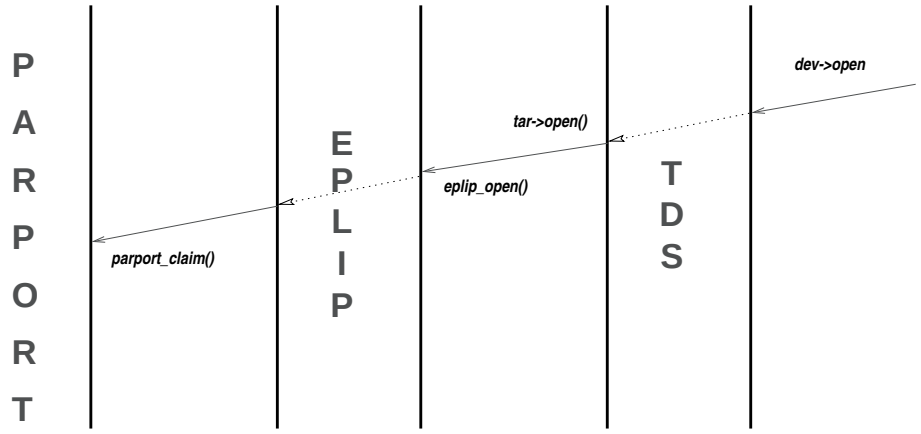

Figura 9.3.: Secuencia de llamadas entre niveles para abrir un dispositivo de red.

```
  dev->type            = ARPHRD\_EPLIP;
  dev->hard\_header\_len = EPLIP\_HARD\_HEADER\_LEN;
  dev->mtu             = EPLIP\_MTU;
  dev->addr\_len         = 0;
  dev->tx\_queue\_len    = 200;   /*   More than in Ethernet       */

  dev->flags           = IFF\_NOARP;
  dev->family          = AF\_INET;
  dev->pa\_addr         = 0;
  dev->pa\_brdaddr      = 0;
  dev->pa\_mask         = 0;
  dev->pa\_alen         = 4;

  dev->tbusy = 1;
  dev->start = 0;

  dev\_init\_buffers(dev);

  return 0;
}
```

Como se ve, es necesario rellenar bastantes campos. Esto es debido a que la estructura `device` del kernel del linux es lo suficientemente genérica para poder ser utilizada para cualquier tipo de adaptador de red, por eso es necesario definir todo lo posible, ya que no se presupone nada.

Por un lado se deben definir las funciones con las que el sistema debe tratar para la gestión de las cabeceras. Estas se detallarán más adelante.

type El tipo de paquete que es. En el caso de `ARPHDR\_EPLIP`, evidentemente tendrá un

identificador nuevo y especial. Las tramas Ethernet tienen otro, las FDDI otro, etc...

hard_header_len La longitud de la cabecera del dispositivo. Tal y como se ha visto en la sección 9.3.1.1 de la página 159 está definida por la macro `EPLIP\_HARD\_HEADER\_LEN`.

mtu *Maximum Transmission Unit* es el tamaño máximo de datos de un paquete (ver sección 2.3. Lo define la macro `EPLIP\_MTU`.

flags Son los indicadores de ciertas propiedades que puede tener un dispositivo de red. Por ejemplo si el dispositivo utiliza ARP...

family Indica el tipo de protocolo que utiliza. En este caso IP.

pa_len Longitud que ocupa una dirección de red del protocolo que utiliza el dispositivo.

Las funciones que se deben definir y implementar para que el nivel de red del kernel del linux pueda operar con el dispositivo son:

change_mtu Esta función la llamará el kernel de red para poder cambiar la *mtu* del dispositivo. Lo único que tendrá que hacer la función es cambiar el tamaño máximo de los paquetes que debe enviar.

hard_header Esta función es llamada cuando el nivel superior debe mandar un paquete y necesita crear su cabecera. Lo que tiene que hacer es rellenar correctamente la cabecera según los valores de los parámetros pasados. De esta manera el nivel superior puede rellenar correctamente las cabeceras de todos los dispositivos solo llamando a las funciones especificas. En el caso de EPLIP, la función deberá reservar espacio al inicio del paquete para la cabecera y rellenar el campo de proto. El código del EPLIP es el siguiente:

```
{
  struct eplip\_hdr *hdr;

  skb\_push(skb,EPLIP\_HARD\_HEADER\_LEN); /* WARNING IS HERE ??? */

  hdr = (struct eplip\_hdr *)skb->data;
  hdr->protocol = htons(type);

  return(EPLIP\_HARD\_HEADER\_LEN);
}
```

rebuild_header También se llama para que se cree correctamente la cabecera del paquete a enviar, pero en el caso que la cabecera sea la de un paquete ARP. Como no es necesario utilizar ARP en EPLIP lo normal es poner un mensaje de error por si se activa accidentalmente.

hard_header_cache Esta función permite a quien la implementa indicar si el tipo de cabecera que se implementa, puede ser reaprovechada. La manera de indicarlo es dando el valor cero al parámetro de retorno. Debido a que las cabeceras de EPLIP incorporan la longitud del paquete a transmitir, no se trabajará con la cache de cabeceras y por tanto la función solo devolverá un valor diferente de 0.

hard_header_cache_update Es la función que permite cambiar la dirección de bajo nivel en la cache de cabeceras. Debido a que no se utiliza la funcionalidad de cache esta función no hace nada ya que no puede ser llamada.

La función `eplip\_close()`, es la inversa a `eplip\_open()`, y se llamará des del nivel superior cuando se quiera acabar con la operación del dispositivo. La mecánica será la misma, llamando a la función `parport\_release()` del nivel inferior.

9.4.2. Envío y recepción de paquetes

Una vez se ha completado la función `eplip\_open()` el puerto paralelo se encuentra en disposición de enviar y recibir paquetes.

Las funciones que define el nivel medio para su funcionamiento son sólo tres:

eplip_write() Es la que utilizará el nivel TDS para enviar los paquetes a un determinado dispositivo de red.

eplip_interrupt() Es la función gestora de la interrupción a la que llamará el puerto paralelo, en la interrupción se produce una negociación del puerto para lograr el control del bus.

eplip_read() Esta función la utilizará el TDS para leer los paquetes que el nivel inferior haya ido reportando mediante la función `prepare\_recv()`.

Debido a que son la base del funcionamiento de este nivel, veamos con más detalle su funcionamiento.

9.4.2.1. eplip_write()

Esta función, almacenada en la estructura `TDS\_target` bajo la función send, recibiría como parámetros el *buffer* y la longitud a transmitir.

El parámetro que se le pasa del nivel superior para indicar por cual de las tarjetas enviar en paquete, es del tipo `TDS\_target`, pero de el se puede extraer la estructura `eplip\_net` a la que corresponde, por medio de su puntero `TDS\_target->priv`.

Lo primero que se hace, a parte de los controles rutinarios de errores, es comprobar si se está realizando una operación de transmisión en curso. Esto se realiza de manera atómica para evitar la corrupción del contador, ya que podría ser modificado simultáneamente por más de un proceso.

```
  pnet=tar->priv;
  if( set\_bit(0,&pnet->inuse) != 0){
    printk("%s: eplip\_write have detected port already in use\n"
           ,tar->dev->name);
    return -1; /* Try latter */
  }
```

Una vez verificado que no se está produciendo ningún tipo de operación sobre el puerto paralelo inicia un proceso de negociación con el host remoto.

```
int request\_to\_send(struct parport *port)
{
  parport\_ieee1284\_reset(port);
  if( parport\_ieee1284\_negotiate(port,0x18) < 0 )
    return -1;
  parport\_switch\_mode(port,PARPORT\_MODE\_ECP);

  return 0;
}
```

Una vez que la negociación se ha realizado exitosamente se llama a la función `parport\_write\_ECP()` [1], indicando los el puerto paralelo, el *buffer*, la longitud.

Una vez enviado el paquete se llama a `finish\_transmition` que activa las interrupciones y cede el control del puerto paralelo si alguien lo ha pedido mediante la función `eplip\_claim`.

9.4.2.2. eplip_interrupt()

Esta función será llamada por el parport para indicar al EPLIP que se ha producido una interrupción, el parport se encarga de ceder el control al EPLIP antes de llamar a `eplip\_interrupt`.

La función `eplip\_interrupt` verifica que el puerto paralelo no está en uso, desactiva las interrupciones del puerto parelelo y empieza la negociación del BUS.

Una vez que la negociación se ha finalizado con éxito lee la cabecera que posee información del tamaño del paquete y del protocolo utilizado.

Al conocer el tamaño del paquete y del protocolo llama a la función `prepare\_recv()` del TDS. Como se ha mencionado en el capítulo 8 en TDS utilizará un thread de kernel y llamará a la función `eplip\_read()` del EPLIP para leer el mensaje que está en curso. No se realiza la lectura en `eplip\_interrupt` pues el puerto paralelo es un dispositivo lento y esta opción bloquearia el kernel durante un tiempo excesivo.

La función `eplip\_read` lee el resto del mensaje y despues finaliza la el mensaje con `finish\_transmition` al igual que ocurren en la función `eplip\_write`.

[1] Actualmente esta función está en el EPLIP, no obstante se está planteando integrarla en el parport para que pueda ser usada por más módulos

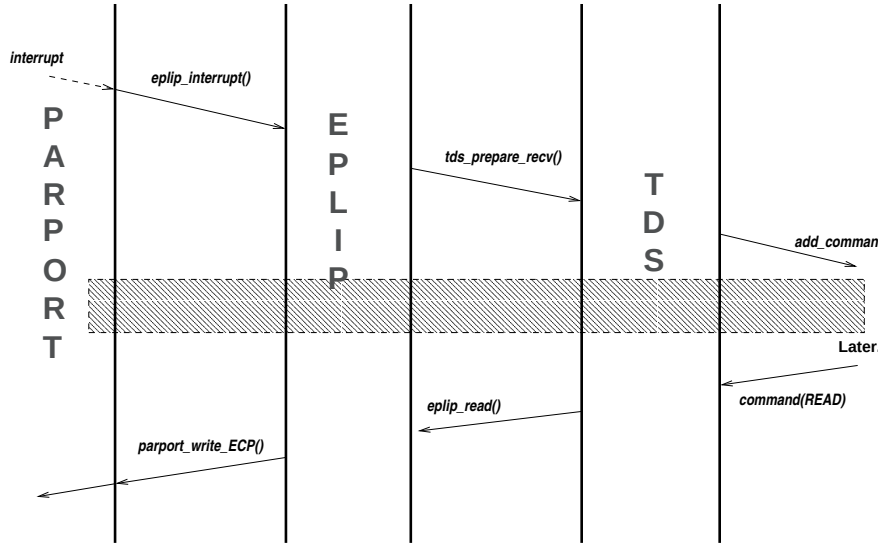

Figura 9.4.: Secuencia de llamadas entre niveles para la recepción de paquetes

9.5. Conclusiones

El modo EPLIP consigue extraer todas las nuevas capacidades del puerto paralelo. Mientras que el PLIP transfiere a 60Kbytes, el EPLIP supera los 200Kbytes. De igual manera se reduce la latencia, pasando de 8ms a 3.2ms en el tiempo gastado en realizar un ping.

Solo por las causas anteriores ya es justificable la implementación del EPLIP, no obstante se ha de añadir que utiliza una nueva negociación compatible con el estándar IEEE1284, por lo que puede operar en daisy chain con otros dispositivos sin causar efectos laterales no deseados.

Capítulo 10 ❑

Conclusiones y valoración

Este capítulo es un análisis desde una perspectiva global del proyecto una vez ha sido diseñado e implementado, con una perspectiva cuando el proyecto está funcionando tal y como se deseaba.

10.1. Evolución

Desde el principio, la discusión y análisis de las distintas alternativas que se iban encontrando ha sido una constante a lo largo del proyecto. El código, a todos los niveles, ha experimentado cambios continuos y optimizaciones. En este proceso se han producido tantos cambios que es casi seguro que no existe ninguna parte en el código actual compartida con las primeras versiones. En proyectos como este, uno recuerda la frase de Descartes: "El descubrimiento del orden no es una tarea fácil...pero una vez se ha encontrado no es difícil reconocerlo".

Esta continua evolución se ha realizado a través de Internet con mailing lists de los. Muy especialmente se ha de agradecer la colaboración de Tim Waugh, Randy Scott, Phillip Blundell, David Campbell y Josep M. Blanquer.

Hay que destacar que para el diseño del proyecto se ha utilizado íntegramente aplicaciones GNU bajo Linux. Ha servido para poder realizar integramente el proyecto sobre esta plataforma, sin necesidad de ningún tipo de software adicional ni de pago. Las utilidades más importantes en el desarrollo del proyecto que han sido utilizadas como parte del sistema operativo, son:

GCC El compilador de C de GNU, utilizado para la compilación del kernel. La ultima versión utilizada es la 2.7.2.1 para la plataforma Intel.

RCS/diff Sistema de control de versiones.

Pine El lector de correo y noticias utilizado para las distintas discusiones y comentarios a través de correo electrónico. La ultima versión utilizada es la 3.95.

Tex/LaTeX El formateador de documentos utilizado para crear la presente memoria. La versión/distribución utilizada es la denominada *tetex*.

Xfig Programa de creación de figuras y dibujos utilizado en la presentación de diagramas y esquemas durante el documento. La ultima versión utilizada es la 3.1.

X/Emacs El entorno de trabajo en la edición de ficheros. Utilizado tanto para la edición de ficheros en lenguaje C, como en los ficheros LaTeX. La ultima versión utilizada es la 20.0.

Durante el desarrollo del proyecto se ha podido constatar que el Linux es uno de los sistemas operativos más competitivos y con más recursos del mercado actual. Existen todo tipo de aplicaciones que permiten realizar todas las fases de un proyecto.

Un ejemplo de ello ha sido el hecho de poder trabajar con *modules* con el consiguiente ahorro de tiempo tanto en compilación como en depuración, ya que no se necesita para el servidor y solo es necesario instalar o desinstalar un módulo del kernel. Esto nos ofrece la ventaja que solo es necesario compilar el módulo sobre el que se trabaja, evitando la compilación de todo el Kernel.

10.2. Las interioridades del núcleo

Cuando se ha de modificar o añadir funcionalidades en el kernel del Linux, hay que tener un buen conocimiento del funcionamiento del núcleo. Es importante conocer la estructura del sistema, pues existen muchas interconexiones y mecanismos más o menos establecidos para su manejo y que se deben conocer profundamente para poder trabajar con ellos.

Debido a ello, se ha de tener en cuenta que el tiempo inicial que necesita una persona para modificar o añadir funcionalidades al kernel del Linux, es bastante alto. Además se debe tener en cuenta que no existe una literatura ni extensa ni por actualizada, debido a los constantes cambios en su estructura.

El desarrollo, cuando se trata de sistema vitales del núcleo, puede llegar a ser lento, ya que aunque pueden usarse *debuggers*, estos no pueden funcionar en muchas partes del Kernel (por ejemplo no se pueden utilizar depuradores de código en las interrupciones o scheduler). De todas formas existen algunos mecanismos de log (`printk()`) y también de *profiling*.

Vistos todos estos aspectos colindantes, se podrían resumir que es un sistema operativo totalmente adecuado para cualquier tipo de proyecto de pequeña o gran envergadura, sobre el cual pueden residir la totalidad de utilidades para desarrollarlo. Por contra se necesita un conocimento profundo de su interior, en caso de que en el proyecto modifique o amplíe el núcleo del sistema.

10.3. El comportamiento del puerto paralelo

El proyecto inicial establecía la necesidad de utilizar como sistema de interconexión placas SCSI y puertos paralelos. Mientras que el comportamiento y experiencia del "mundo" SCSI ha sido altamente satisfactoria y solo se han presentado problemas en

la parte de prestaciones. En el puerto paralelo ha sucedido lo contrario. El hardware es muy especial, con multitud de particularidades según el fabricante del chip y versión de protocolo soportado. No obstante se han conseguido resultados más elevados de lo esperado inicialmente.

Tal y como ha sucedido en este proyecto, mientras se desarrolla un proyecto en Linux, puede surgir la necesidad de ampliar alguna otra parte del Kernel. Esta situación ocurrió al implementar el puerto paralelo, pues se pudo observar que seria aconsejable un cambio de filosofía en el funcionamiento del Kernel, incorporando el nuevo módulo (parport) en la gestión de los puertos paralelos.

10.4. Resultados preliminares

Realizada la implementación, aunque se debe pensar en que debe seguir evolucionando y mejorando en el máximo de aspectos, se pueden extraer los primeros resultados de su comportamiento, eficiencia y eficacia.

Para los primeros resultados, se utilizarán programas habituales de transmisión sobre TCP/IP, y algunos pequeñas utilidades orientadas a medir tiempos y anchos de banda de sistemas de interconexión. En esta serie de pruebas preliminares no están incluidos tests intensivos con variaciones de los parámetros más importantes, ni grandes aplicaciones de uso intensivo de las redes de interconexión como podría ser una aplicación paralela sobre PVM (*Parallel Virtual Machine*), tests sobre otros protocolos diferentes al TCP/IP, transmisiones *raw* directas al nivel bajo de transmisión ... y se dejan como un futuro anexo. El motivo de que no estén presentes en un capítulo separado, se debe a que hay temas como la posible aceptación del subsistema por la comunidad de Linux, o la incorporación de nuevos módulos de tipos de interconexión diferentes, que permitirían una mejor consolidación en el sistema global.

Los diferentes tests preliminares, se pueden separar fácilmente según el tipo de característica se quiera medir. Las siguientes secciones tratan de las características que se han considerado más importantes.

10.4.1. Latencias

Una característica básica necesaria para medir el rendimiento de un sistema de interconexión, es la latencia que tiene entre diferentes nodos. La latencia se entiende como la medida de tiempo que tarda un "paquete" o unidad de información, desde el momento que es enviado hasta que es recibido por su destinatario.

Dependiendo del tipo de datos que estemos enviando, será más o menos importante que el sistema tenga latencias cortas o largas. No obstante siempre es mejor una latencia corta que otra larga.

El proyecto se ha realizado usando placas SCSI y puertos paralelos, a continuación se comentan las latencias que se han encontrado.

En el caso de la controladora SCSI utilizada, la latencia es un factor determinantemente malo para su eficiencia. Así pues, la latencia mínima medida entre dos tarjetas del

tipo aha1542 que estén conectadas a un mismo bus SCSI, se aproxima a los 3,5ms. Es decir, este es el tiempo que transcurre desde que se programa una transferencia de datos a la controladora, hasta que esta nos informa de que ya han sido todos transmitidos (y por tanto el *target* los ha recibido).

Esta latencia es muy elevada si la comparamos con una Ethernet de 10Mb/s que esta alrededor de los 0,5ms. Las causas de esta latencia, son debidas a las rutinas que implementan los chips internos que se encargan del establecimiento de la conexión, negociación del bus, controlar las diferentes fases y manejar las colas de comandos entrantes y salientes de entrada. En el caso que la controladora pudiese soportar más frecuencia de reloj u otro modelo de procesador seguramente esta latencia se reduciría drásticamente. Por ejemplo en placas NCR, Tim Waugh comenta que se consiguen latencias entre 1 y 2 milisegundos.

El puerto paralelo posee unas latencias bastante más reducidas. Por ejemplo para realizar un round-trip entre de un byte entre dos máquinas solo es necesario 40us. No obstante si es necesario realizar una negociación de bus, como es el caso del TCP/IP, el tiempo aumenta a 1ms.

Se puede observar que en relación con el módulo PLIP, el EPLIP ha supuesto una reducción en la latencia del 50 %.

Tipo	Ethernet 10Mb	localhost	SNIP	EPLIP	PLIP
TCP	758us	236u	7808u	3000us	6000us
UDP	686us	189us	7804us	2800us	5800us
RPC	934us	365us	7847us	3200us	6400us

Cuadro 10.1.: Comparativa de latencias desde diferentes niveles de red

Los programas utilizados para realizar los tests mostrados en la tabla 10.1, han estado extraidos de la suit de pruebas de *Larry McVoy* [MS96].

10.4.2. Bandwidth

Otra de las características fundamentales para evaluar las redes de interconexión es el ancho de banda que tiene. El concepto de ancho de *bandwidth* significa la cantidad de información que se puede transmitir por unidad de tiempo.

En ocasiones, dependiendo de la aplicación, es más importante el ancho de banda que la latencia, aunque a veces prepondera la latencia que el ancho de banda.

Los benchmarks se han realizado utilizando una de las herramientas que proporciona la suite de *benchmarking* de Larry McVoy [MS96].

La comparativa entre las mismas máquinas, de los índices que da el programa `bw_tcp` anteriormente citado, se recoge en la tabla 10.2.

Como se ve en la tabla 10.2 existe una gran diferencia entre el dispositivo del localhost, las plataformas ethernet, SCSI, PLIP y EPLIP. Esto es debido a que el dispositivo localhost simula un dispositivo de red para consigo mismo, de manera que cualquier

Plataforma	*Bandwidth*
localhost	$16700KB/s$
ethernet	$989KB/s$
SNIP	$670KB/s$
EPLIP	$220KB/s$
PLIP	$65KB/s$

Cuadro 10.2.: Comparativa de *bandwidth* con *bw_tcp*.

paquete que se le envíe, el ancho de banda está limitado a la velocidad de copia de memoria a memoria. En el caso del Linux, este coeficiente es de $\frac{1}{2}$ ya que ha de realizar dos copias de memoria.

En cuanto a la plataforma SCSI, se debe decir el ancho de banda conseguido esta aún un poco lejos de los resultados que se esperan de una buena controladora. La causa de su bajo índice se deriva de una combinación del problema de las largas latencias.

Mientras el EPLIP consigue aumentar el ancho de banda en un 400 % con respecto con el PLIP, lo cual es bastante aceptable.

10.5. Conclusiones del proyecto

Durante el desarrollo del proyecto se han extraido múltiples conclusiones, estas observaciones abarcan desde áreas arquitectónicas, hasta técnicas de trabajo en grupo o gestión de proyectos de gran volumen.

Algunas conclusiones referentes a problemas arquitectónicos de los PCs actuales son:

- La problemática que presenta ciertas compatibilidades con ordenadores que se diseñaron hace más de 10 años.
- Una limitación de las arquitecturas actuales se presenta en la velocidad de copiar de memoria a memoria, por ejemplo en un i386 solo se copia a 3.5Mbytes/s, mientras que en un Pentium solo se alcanzan valores de 30Mbytes/s.
- En los protocolos de red analizados se puede observar que el TCP/IP no es apropiado para aplicaciones paralelas, el problema no reside en el overhead requerido para la gestión de los paquetes, sino en la necesidad de realizar dos copias de memoria cada vez que se transmite un paquete. Se ha de observar que este problema no puede solucionarse a causa de las características del TCP/IP [1].

Con la intención de solucionar los problemas de latencia y ancho de banda que representa el protocolo TCP/IP se puede recurrir a arquitecturas como la propuesta por el SCI (CC-NUMA) o utilizar active message [2].

Se ha podido constatar la existencia de buena voluntad entre la comunidad Linux a la hora de soportar nuevas características en el Kernel. Durante el diseño e integración del `parport` existió una alta compenetración enviando varios mensajes diariamente.

Gracias a la creación del proyecto `parport`, desde la versión 2.1.33, se puede conectar varios dispositivos en cascada en un mismo puerto paralelo. Por ejemplo es posible conectar una unidad ZIP y una impresora en un solo puerto paralelo y utilizar los dos simultaneamente. Se ha posibilitado la utilización de los nuevos modos del puerto paralelo, con el consiguiente aumento de prestaciones, y también se ha facilitado una API común para la utilización en otras arquitecturas como máquinas SUN o PPC.

[1] Solo en dispositivos localhost podría solucionarse y conseguir realizar todo el proceso con una sola copia de memoria

[2] En los active message se evita una copia entre memorias

Parte IV.

Apéndices

Capítulo 11 ❑

The GNU General Public License

Printed below is the GNU General Public License (the *GPL* or *copyleft*), under which Linux is licensed. It is reproduced here to clear up some of the confusion about Linux's copyright status— Linux is *not* shareware, and it is *not* in the public domain. The bulk of the Linux kernel is copyright © 1993 by Linus Torvalds, and other software and parts of the kernel are copyrighted by their authors. Thus, Linux *is* copyrighted, however, you may redistribute it under the terms of the GPL printed below.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

11.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

11.2. Terms and Conditions for Copying, Distribution, and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

    b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

    c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

    These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

    Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

    In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may

not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does

not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

11.3. Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.* Copyright © 19yy *name of author*
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items— whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Bibliografía

[Ada] Adaptec. *Adaptec AHA-1542 Programming Guide.*

[Alma] Werner Almesberger. *LILO Generic Boot Loader for Linux, Technical Overview.* Version 1.9.

[Almb] Werner Almesberger. *LILO Generic Boot Loader for Linux, User's Guide.* Version 1.9.

[And97] Don Anderson. *Universal Serial Bus System Architecture.* Addison-Wesley Developers Press, 1997.

[Axe96] Jan Axelson. Answers to some frequently asked questions about the parallel port. Technical report, lvr, 1996.

[Axe97] Jan Axelson. *Parallel Port Complete.* Lakeview Research, 1997.

[Bak96] Art Baker. *The Windows NT Device Driver Book.* Prentice Hall, 1996.

[Bar93] Robert Baruch. Tutorial to linux driver writing – character devices. Also known as The Wacky World of Driver Development I, April 1993.

[BBD+96] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals.* Addison-Wesley, 1996.

[BG93] Mats Björkman and Per Gunningberg. Lockign effects in multiprocessor implementations of protocols. In *SIGCOM93.* SIGCOM93, 1993.

[BL95] David B.Gustavson and Qiang Li. Local-area multiprocessort: the scalable coherent interface. Technical report, Santa Clara University, 1995.

[Cir] Cirrus Logic. *CL-CD1283: IEEE1284-Compatible Parallel Interface.* IEEE1284 compatible chip with 64Bytes in FIFO.

[CO96] John P: Choisser and John O.Foster. The pc handbook, 1996.

[com86] ANSI SCSI comitee. *ANSI SCSI-1 Standard X3.131-1986*, 1986.

[com94] ANSI SCSI comitee. *ANSI SCSI-2 Standard X3.131-1994*, 1994.

[Com95] Compaq and Phoenix and Intel. *BIOS Boot Specification*, 1995. BIOS Boot Specification.

[Cor93a] Microsoft Corporation. Ecp compliance test functional specification. Technical report, Microsoft Corporation, 1993.

[Cor93b] Microsoft Corporation. Extended capabiblities port specification. Technical report, Microsoft Corporation, 1993.

[Daw96] Terry Dawson. Linux networking 2 howto. Version v3.5, January 1996.

[DTM95] H.G. Diets, T.M. Chung T.Mattox, and T. Muhammad. Pardue's adapter for parallel execution and rapid synchronization: The ttl_papers design. Technical report, Pardue University, 1995.

[Eib94] Heiko Eibfeldt. The linux scsi programming howto. This document deals with programming the Linux generic SCSI interface, 1994.

[FAPa] FAPO. Epp mode. http://www.fapo.com.

[FAPb] FAPO. Nibble mode. http://www.fapo.com.

[H.A95] Peter H.Anderson. Use of a pc printer port for control and data acquisition. Technical report, Morgan State University, 1995.

[HAC+95] H.Müller, A.Bogaerts, C.Fernandez, L.McCulloch, and P.Werner. A pci-sci bridge for high rate data acquisition architecture at lhc. Technical report, CERN, 1995.

[Hex94] Roberto A. Hexsel. *A Quantitative Perdormance Evaluation of SCI Memory Hierachies*. PhD thesis, University of Edinburgh, 1994.

[Hol] Holtek. *HT6535 SPP/EPP/ECP Controller*. January 1996.

[Hom84] C. Homming. *Ethernet Address Resolution Protocol*, 1984.

[IEE95a] IEEE. Sci/rt scalable coherent interface for real-time applications. Technical report, IEEE, 1995.

[IEE95b] IEEE. Stardard for the interface and protocol extensions to ieee std1284-1994 compliant peripheral and host adapter ports. Technical report, IEEE, 1995.

[Int95] Intel. 82091aa: Parallel port operations. Technical report, Intel Corporation, 1995.

[Int96] Texas Intruments. Tl16pir552 dual uart with dual irda and 1284 parallel port. Technical report, Texas Intruments, 1996.

[JM90] S.E. Deering J.C. Mogul. *Path MTU discovery*, 1990.

[J.P80] J.Postel. *User Datagram Protocol*, 1980.

[J.P81] J.Postel. *Internet Protocol*, 1981.

[JR88] J.Postel and J.K. Reynolds. *Ethernet Address Resolution Protocol*, 1988.

[Kir94] Olaf Kirch. The linux network administrators guide. part of the LDP, 1994.

[LHFL95] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law. In-kernel servers on mach 3.0: Implementation and performance. Technical report, University of Utah, 1995.

[LRK96] Steven L.Scott, James R.Goodman, and Mary K.Vermon. Performance of the sci ring. Technical report, University of Winsconsin, 1996.

[MBJS96] Marshall Kirk McKusick, Keith Bostic, Michael J.Karels, and John S.Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[McG92] G. McGregor. *The PPP Internet Protocol Control Protocol (IPCP)*, 1992.

[Mic] Michael K: Johnson. *Linux Kernel hackers Guide*.

[Mic94] Microsoft and Intel. *Plug And Play ISA Specification*, 1994. Plug And Play ISA Specification, version 1.0a.

[Mic96] Microsoft and Intel. *Plug And Play Parallel Port Devices*, 1996. Plug And Play Paralle Port Devices, version 1.0a.

[MS96] Larry McVoy and Carl Staelin. lmbench - portable tools for performance analysis. Technical report, USENIX, 1996.

[Nat] National Semiconductor. *PC87322VF (SuperI/O III) Ploppy Disk Controller with Dual UART Enhanced Parallel Port, and IDE Interface*. Preliminary June 1994.

[Nel91] Mark Nelson. *The Data Compression Book*. Prentice Hall, 1991.

[Plu82] D.C. Plummer. *Ethernet Address Resolution Protocol*, 1982.

[Rad96] Paul Rademacher. Programming the enhanced parallel port. Technical report, www, 1996.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley Professional Computing Series, 1994.

[Sch95] Friedhelm Schmidt. *The SCSI bus and IDE interface, protocols applications and programming*. Addison Wesley, 1995.

[SFPB95] Emin Gün Sirer, Marc E. Fiucynski, Przemyslaw Pardyak, and BrianÑ. Bershad. Safe dynamic linking in an extensible operating system. Technical report, University of Washington, 1995.

[Sim94] W. Simpson. *PPP LCP Extensions*, 1994.

[SMC] SMC. *FDC37C93xFR.* SMC Plug and Play Compatible Ultra I/O Controler with Fast IR.

[S.T91] Andrew S.Tanembaum. *Redes de ordenadores.* Prentice Hall, 1991.

[Sta96] Startech. Ecp/epp parallel printer port with 16 byte fifo. Technical report, Startech, 1996.

[Ste94] Zahai Stewart. Interfacing the ibm pc parallel printer port. Technical report, mri, 1994.

[Ste95] W. Richard Stevens. *TCP/IP Ilustrated, Volume 1: The Protocols.* Addison-Wesley Professional Computing Series, 1995.